NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93943-5101

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (*If applicable*) 55	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000	7b. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*)	10. SOURCE OF FUNDING NUMBERS Program Element No. / Project No. / Task No. / Work Unit Accession Number

11. TITLE (*Include Security Classification*)
Re-engineering Software Systems in the Department of Defense Using Integrated-Computer Aided Software Engineering Tools

12. PERSONAL AUTHOR(S) Jennings, Charles A.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*year, month, day*)	15. PAGE COUNT
Master's Thesis	From To	92 September 24	124

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18. SUBJECT TERMS (*continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUBGROUP	Software Re-engineering, Reverse Engineering, Computer-Aided Software Engineering, Integrated -Computer Aided Software Engineering

19. ABSTRACT (*continue on reverse if necessary and identify by block number*)

The Department of Defense (DoD) is plagued by severe cost overruns and delays in developing software systems. Existing software within DoD, some developed 15 to 20 years ago, require continual maintenance and modification. Major difficulties arise with maintaining older systems due to cryptic source code and a lack of adequate documemtation. To remedy this situation, the DoD, is pursuing the integrated computer aided software engineering (I-CASE) procurement as a means to improve DoD's development and maintenance of software systems. This study focuses on the concepts and theory behind software re-engineering. In particular, it studies the current state of I-CASE technology, and the feasibility of re-engineering existing software systems for migration to an I-CASE environment.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS REPORT ☐ DTIC USERS	Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL Martin J. McCaffrey	22b. TELEPHONE (*Include Area code*) (408) 646-2488	22c. OFFICE SYMBOL AS/Mf

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

T260868

Approved for public release; distribution is unlimited.

Re-Engineering Software Systems In The Department Of Defense Using Integrated Computer Aided Software Engineering Tools

by

Charles A. Jennings
Lieutenant, United States Navy
B.S.B.A. University of Alabama in Huntsville, 1982

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
September 1992

ABSTRACT

The Department of Defense (DoD) is plagued by severe cost overruns and delays in developing software systems. Existing software within DoD, some developed 15 to 20 years ago, require continual maintenance and modification. Major difficulties arise with maintaining older systems due to cryptic source code and a lack of adequate documentation. To remedy this situation, the DoD, is pursuing the integrated computer aided software engineering (I-CASE) procurement as a means to improve DoD's development and maintenance of software systems. This study focuses on the concepts and theory behind software re-engineering. In particular, it studies the current state of I-CASE technology, and the feasibility of re-engineering existing software systems for migration to an I-CASE environment.

J406
C.1

THESIS DISCLAIMER

The following trademarks are used throughout this thesis:

AD/Cycle	is a registered trademark of International Business Machines Corporation.
Application Development Workbench	is a trademark of Knowledgeware, Inc.
Bachman/Analyst and Bachman/Database Administrator	are trademarks of Bachman Information Systems, Inc.
Battlemap Analysis Tool	is a registered trademark of McCabe & Associates, Inc.
CDD/Repository	is a registered trademark of Digital Equipment Corporation.
Cohesion	is a registered trademark of Digital Equipment Corporation.
DB2	is a registered trademark of International Business Machines Corporation.
IBM	is a registered trademark of International Business Machines Corporation.
IEF	is a registered trademark of Texas Instruments Inc.
IMS	is a registered trademark of International Business Machines Corporation.
Inspector	is a trademark of Knowledgeware, Inc.
Navigator Systems Series	is a service mark of Ernst & Young.

Pinpoint	is a trademark of Knowledgeware, Inc.
Rapid Application Development	is a trademark of James Martin Associates.
Recorder	is a trademark of Knowledgeware, Inc.
RE/Toolset	is a trade mark of Ernst & Young.
Repository Manager/MVS	is a registered trademark of International Business Machines Corporation.

TABLE OF CONTENTS

I. INTRODUCTION 1

A. BACKGROUND 1

B. RESEARCH QUESTIONS 2

C. METHODOLOGY 3

D. FOCUS 3

E. ORGANIZATION OF THE THESIS 4

II. RE-ENGINEERING AND REVERSE ENGINEERING 6

A. OVERVIEW 6

B. RE-ENGINEERING POTENTIAL 21

C. RE-ENGINEERING LIMITATIONS 31

D. A TAXONOMY OF A RE-ENGINEERING PROJECT 35

E. SUMMARY 39

III. INTEGRATED COMPUTER-AIDED SOFTWARE ENGINEERING . . .41

A. A DEFINITION OF INTEGRATION 42

B. INTEGRATION SUBCOMPONENTS 44

C. CASE VERSUS I-CASE 48

D I-CASE REPOSITORY 49

E. METHODOLOGY 54

F. INTEGRATED ARCHITECTURE 59

G. I-CASE BENEFITS 60

H. I-CASE LIMITATIONS 61

I. THE DoD I-CASE PROCUREMENT 63

J. SUMMARY 65

IV. RE-ENGINEERING WITH I-CASE IN DoD: DATA COLLECTION 67

A. DATA SEARCH 67

B. INQUIRY BACKGROUND 68

C. INFORMATION SOLICITED 68

D. DATA RESULTS 69

E. SUMMARY 72

V. CONCLUSIONS AND RECOMMENDATIONS 73

A. CONCLUSIONS 73

B. ANSWERS TO RESEARCH QUESTIONS 75

C. LESSONS LEARNED 80

D. FINAL THOUGHTS AND RECOMMENDATIONS 81

APPENDIX A: THE RE-ENGINEERING CANDIDATE SELECTION SELECTION PROCESS 84

APPENDIX B: RE-ENGINEERING QUESTIONNAIRE 96

APPENDIX C: LIST OF FIGURES 100

APPENDIX D: LIST OF TABLES 106

LIST OF REFERENCES 110

INITIAL DISTRIBUTION LIST 115

I. INTRODUCTION

A. BACKGROUND

In the 1960's, up to 80% of the cost of a computer system were attributed to hardware and 20% to software. [Ref. 38:p. 1] By 1985 this trend had dramatically reversed; software maintenance consumed as much as 80% of system budgets. [Ref. 38:p. 1] This change was facilitated by the increased processing power and reduced cost of hardware. The increased cost of software was due primarily to growth in program size, increasing complexity of programs, and an ever growing software maintenance pool. The increased emphasis on software costs mandated that software be developed not only for initial functionality, but also to have characteristics that would enable cost effective maintenance.

Over the years, the DoD has witnessed its software inventory grow by millions of lines of code. Much of it being old (legacy) code. Many of these older DoD software systems, created prior to the implementation of structured methodologies, were developed along artistic "ad hoc" means. Worse than the poor design techniques used for this software was the frequent lack of adequate software documentation. The cost to maintain old systems are enormous. Progress in operating systems, microprocessor capability, and telecommunications has enabled faster, more flexible, and less

costly computing power than ever before. Many obsolete systems are being up-graded. Many government organizations have found that older and functional systems are still quite useful when migrated to newer computing platforms. Two years ago DoD introduced the Corporate Information Management (CIM) initiative to stream-line and reduce the costs of its information technology. One of the goals of CIM is to capitalize on Integrated Computer Aided Software Engineering (I-CASE) and re-engineering technology to develop new systems and re-engineer existing systems that will provide improved, more cost efficient systems. This thesis will focus on these issues.

B. RESEARCH QUESTIONS

This thesis will focus on the following research questions.

1. Primary Question

From DoD's standpoint, what needs to be considered, as well as avoided, in re-engineering its inventory of systems within an Integrated Computer Aided Software Engineering (I-CASE) environment?

2. Subsidiary Questions

a. What are the current problems facing the CASE and I-CASE industry?

b. Can re-engineering using I-CASE tools produce viable systems for DoD?

c. How many systems within DoD warrant re-engineering?

d. What are the estimated cost savings DoD can anticipate by re-engineering some of its applications?

C. METHODOLOGY

This research was developed in four stages. First, a literature review was conducted on CASE, I-CASE, and software re-engineering. This set the ground work for understanding the theory and attributes of current CASE/I-CASE technology as well as the work that had been completed in these domains. Second, interviews and site visits to CASE/I-CASE vendors, and attendance at recent CASE and re-engineering conferences enabled the preliminary collection of data. The third stage of research consisted of telephone interviews and electronic mail (e-mail) correspondence with government, industry, and academic personnel. It further enhanced the understanding of re-engineering and I-CASE technology issues. Finally, a questionnaire was developed and sent to three government locations actually working with I-CASE tools. The response to the questionnaire was analyzed and served, along with information gathered from other sources, as the basis for the conclusions and recommendations at the end of the thesis.

D. FOCUS

This research was designed to collect information on organizations' use of I-CASE tools for software re-engineering. Initially, the major objectives were the following:

1. analyze the benefits of using an I-CASE tool for re-engineering;

2. access the learning curve, i.e., how did personnel adjust to using an I-CASE tool, and how long did it take to become proficient in the use of an I-CASE tool;

3. determine what the lessons learned were from re-engineering with an I-CASE tool.

However, due to the time constraints involved with this research, plus limited available data from organizations' using I-CASE tools in a re-engineering capacity, a large sample of data was not obtainable that would have helped in analyzing the items listed above. Instead, the focus of this research shifted to explaining the theory and managerial issues surrounding software re-engineering and I-CASE. One organization was found using an I-CASE tool in a re-engineering capacity. However, the data obtained was not sufficient to lead to any substantial conclusions on benefits, learning curves, or lessons learned.

E. ORGANIZATION OF THE THESIS

This thesis is organized into five chapters. Chapter II presents an overview of the theory of re-engineering. It discusses technical and managerial considerations involved with a software re-engineering process, plus capabilities and limitations of re-engineering. Chapter III reviews the components and theory involved with I-CASE along with I-CASE benefits and limitations. Chapter IV covers data collection and the results of the research findings. Chapter V concludes

with the lessons learned from the research and offers recommendations for future research.

II. RE-ENGINEERING AND REVERSE ENGINEERING

A. OVERVIEW

This chapter provides an overview of software reverse engineering and re-engineering. The differences between the two processes and their relationship in terms of the systems development life cycle (SDLC) and Computer-Aided Software Engineering (CASE) tools are discussed. Particular attention will be placed on key terminology and definitions associated with re-engineering. Capabilities and limitations associated with re-engineering, as well as selection criterion for re-engineering projects, are discussed. The chapter concludes with a discussion of steps comprising a successful re-engineering project.

1. Definitions

Re-engineering is a relatively new and emerging technology. Thus, the definition of the terms re-engineering and reverse engineering may vary depending on the source. One widely accepted definition of the terms re-engineering, reverse engineering, and forward engineering are as follows:

> Re-engineering, also known as renovation and reclamation, is the examination and alteration of a subject system to reconstitute it in a new form, and the subsequent implementation of the new form. [Ref. 1:pp. 15-16]

> Reverse engineering is the process of analyzing a subject system to identify the system's components and their inter-relationships and create representations of the system in another form or at a higher level of abstraction. [Ref. 1:p. 15]

> Forward Engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent design to the physical implementation of a system. [Ref. 1:p. 14]

The previous definitions are fairly formal and may appear cumbersome, especially if one is not familiar with re-engineering technology. So perhaps a different, yet easily understandable definition to these re-engineering terms is warranted:

> Software re-engineering. A combination of tools and techniques that facilitate the analysis, improvement, redesign and reuse of existing software systems to support changing information needs. [Ref. 42:p. U-2]

Reverse engineering is the analysis of an existing system in order to represent it in another form. For example, logical data models, data flow diagrams, entity-relationship diagrams, or action diagrams could be selected as other forms of representation. [Ref. 20:p. 2]

Forward engineering can be considered the process of developing a system from definition/analysis through design, to code construction and testing, to the eventual implementation and acceptance of a working system. It includes testing, documentation and configuration management.

A more pragmatic view of defining re-engineering is in the area of reuse. Re-engineering is not software reuse per se, but rather a means of facilitating reuse. In broad terms, software reuse is taking a segment of code from one system and transporting (i.e., reusing) it to another program without

modification. It still functions as designed. For instance an algorithm that computes a radio frequency may be used as an example. The ability of the algorithm to work in different programs is dependent upon the programming language and operating system that the algorithm was originally created in. If different operating environments are compatible, then the algorithm could be used. Re-engineering's application of reuse differs in the sense that it uses existing code that is then modified by using a set of techniques, tools, and methodologies. [Ref. 39:p. 3] More efficient, effective and maintainable software is the result.

2. Relationship between Re-engineering and Reverse Engineering

"Re-engineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some form of forward engineering or restructuring." [Ref. 1:p. 15] In other words, the relationship between re-engineering and reverse engineering could be stated as follows:

Re-engineering = Reverse Engineering + Forward Engineering.

This definition tends to be the one most broadly accepted by the software industry. [Ref. 6:p. 1] By transforming the program code into higher levels of abstraction, the business rules[1] and characteristics of the

---

[1] Business rules or business intentions of a system are the values or constraints that the source code embodies. For example: Gross Pay = Hourly Rate * Hours Worked [Ref. 5:p. B6-18].

code can be further examined. [Ref. 1:p. 15] To lay the ground work for understanding re-engineering, one must first comprehend the concept of abstractions: how they are created and their function in the reverse engineering process. Both items are discussed in the following section.

3. Reverse Engineering and the Concept of Abstraction

Understanding the concept of abstraction is the key point to grasping the re-engineering process. Raising program code to higher levels of abstraction is associated with reverse engineering. This is because the reverse engineering process takes existing program source code, which can be thought of as a physical description of a system, or more commonly "how" a system works, and transforms the source code to a specification level. The specification level describes "what" the system code does. [Ref. 9:pp. 54-55] An analogy of abstraction would be that of a simple road map. A map can show you how to travel from point A to point B without displaying every bend in the road between the two points. In terms of software engineering terminology, an abstraction may be in the form of data flow diagrams (DFD's), or entity-relationship diagrams (ERD's) that serve as abstractions representing the program source code. Two key benefits of expressing source code in a higher level of abstraction are:

a. Size: fewer lines of code are needed to represent the original source code when it is represented in a process module of a DFD or ERD. [Ref. 5:p. B6-6]

b. Context: source code in a higher level of abstraction is more context free. For example, in human speech words are often constrained and defined depending on they are used in a sentence. In the declarative phrase "Look! I said to move these items now, not tomorrow." the exclamation mark after the word "look" conveys attention to the order that is about to follow. Whether this is written or spoken, a human can understand the semantics associated with the command. But if the same phase is changed slightly to "Excuse me. Please move these items now, not tomorrow," the meaning is altered. Higher level abstractions are more context free in the sense that the meaning contained in them is not lost or altered because of their placement or relationship with other abstraction modules. [Ref. 5]

There are limitations to the extent to which source code may be represented at higher levels of abstraction. Further discussion into these limitations are discussed later in this chapter under the topic of technical considerations. However, the salient feature of reverse engineering is that design elements are created and stored into a repository.[2] This data in the repository will be used in the forward engineering process, which is covered in the following section.

There are numerous reverse engineering tools available. The following is a brief classification of reverse engineering tools and examples from various vendors:

a. Migrating tools translate from one language or database to another. For example from COBOL to C, or Information Management System (IMS), Database 2 (DB2) to another database system. [Ref. 20:p. 3] Products include Bachman/Analyst

[2] Repositories are covered in greater detail in Chapter III.

and Bachman/Database Administrator from Bachman Information Systems Inc. [Ref. 43:p. 73]

b. Restructuring tools can scan old COBOL, C, Ada, Fortran, Pascal, PL/1, or Assembler code and make them easier to read, i.e., more structured. [Ref. 20:p. 3] Products for COBOL include Application Browser and Hypercode Management System from Hypersoft Corporation. [Ref. 43:p. 73]

c. Design recovery tools extract business rules from existing code. [Ref. 20:p. 3] Products include RE/Toolset from Ernst & Young and InterCASE from Interport Software Corporation.

d. Static logic analyzers[3] read code and prepare graphic representations of its logic and control flows. They help pinpoint potential side effects of a code change, and provide up-to-date documentation of a system. [Ref. 20:p. 3]

e. Complexity tools enable programmers and maintainers of software systems to visualize the structure of a program by creating on-screen graphs. It is practically impossible to determine the structure of a program by manually reading source code line by line. Complexity tools will identify which sections of a program are unmaintainable and untestable. One of the better known complexity tools is the Battle Map Analysis Tool from McCabe & Associates, Inc.

Since the reverse engineering process provides a closer examination of data at a higher abstraction level, it allows for two important capabilities:

a. A better understanding of the current system's complexity and functionality, which enables the identification of "trouble spots" within a program. [Ref. 38:p. 4]

---

[3] A thorough report that lists and analyses various static analysis tools can be found in "Source Code Static Analysis Tools Report, April 1992," published by the Software Technology Support Center, Hill Air Force Base, UT 84056.

b. A means to restructure or "clean up" data in order to forward engineer and change it into a new system. [Ref. 2:p. 4]

4. Forward Engineering

The forward engineering process picks up where the reverse engineering process left off. The design elements created and stored in the repository during the reverse engineering phase are now available for manipulation by CASE or I-CASE tools. Theoretically this process sounds simple and intuitive, but in actuality the forward engineering process can be very time consuming. Forward engineering requires developers to analyze, plan, construct and test code. CASE and I-CASE tools help the developer in automating many of the items that were once manual processes. For example, design elements in a repository provide the data (files, entities, relationships among data etc.), but people are needed to place the data in a coherent structure that, when manipulated by a CASE or I-CASE tool, results in a functional system. There are numerous CASE tools available for forward engineering. Each have both similar and different capabilities.[4]

5. Distinction between Re-engineering and Reverse Engineering

In order to understand the re-engineering process, it is fundamental to comprehend the distinction between reverse

---

[4] A thorough report that covers forward engineering tools is "Re-engineering Tools Report March 1992," published by the Software Technology Support Center, Hill Air Force Base, UT 84056.

engineering and re-engineering. The distinction is thus: reverse engineering does not change a system, it only extracts data to a higher level of abstraction. Re-engineering takes the information derived from the reverse engineering process and forward engineers the information, using CASE or I-CASE tools, into a into a new form, but without changing the function of the system. [Ref. 5:p. B6-1]

Figure 2-1 in Appendix C, displays a simple diagram of reverse engineering and re-engineering.[5] The gist of the diagram is to show the distinction between reverse engineering and re-engineering. Two key points of Figure 2-1 are [Ref. 5]:

a. Reverse engineering only raises program code to a higher specification level of abstraction, this is represented by the arrow moving from the code to the specification level. The reverse engineering process does not change a system.

b. Re-engineering takes the higher level specification abstractions and changes the abstractions into new code that has similar functionality to that of the original source code. This is represented by the arrow first moving from the code to the specification level then returning back down to the code. For example, a payroll system that is written in COBOL is characterized as being unstructured (i.e., difficult to read, understand and maintain). Re-engineering will take the information contained from the reverse engineering process and make the program more structured and thus maintainable.

---

[5] This diagram was presented by Dr. Eric Bush on February 18, 1992 at the CASE World Conference & Exposition held in Santa Clara, California, and again on August 11, 1992 at the National Software Re-engineering and Maintenance conference held in San Jose, California.

6. Systems Development Life Cycle

During the 1980's the systems development life cycle (SDLC) methodology became a traditional means of implementing a computer system. There are many versions and definitions for SDLC. A basic, yet thorough definition is as follows:

> A systems development life cycle (SDLC) is a process by which systems analysts, software engineers, and programmers build systems. It is a project management tool, used to plan, execute, and control systems development projects. [Ref. 21:p. 81]

As shown in Figure 2-2, the SDLC is composed of four major phases: systems analysis, systems design, systems implementation, and system maintenance/support. The SDLC methodology is also used for software re-engineering. The following terms are frequently associated with the SDLC:

a. Feasibility Study. This process determines whether or not significant resources should be committed to the other phases of the SDLC. The feasibility study will also define the scope of a project, perceived problems and opportunities, business and technical constraints, perceived project goals and possible solutions. [Ref. 21:p. 87]

b. System Analysis. After the feasibility study has identified a need, analysis is conducted to identify the current capabilities of an existing system. You cannot enhance a system without first understanding "how" it works. The analysis phase must produce a specification that outlines the functional and data requirements of a system.

c. System Design. Data flow diagrams, entity relationship diagrams, file layouts and other structured techniques designed to represent and capture data in different abstraction levels are accomplished during this phase. abstractions of program code created during the design phase are language, operating system and database management system (DBMS) independent. No coding

is done during the design phase.

d. System Implementation. The actual programming of code for the complete system is accomplished during this phase.

e. Testing. Consists of internal testing of the system prior to delivery to the user. The internal tests can be categorized as unit tests, integration tests, and performance tests. Unit testing focuses on the smallest unit of software design, which is the module. Integration testing is conducted to uncover any errors that may occur by bringing different modules together under one system. Performance testing is conducted after integration testing. Performance testing[6] consists of various types of tests, notably stress testing. Stress testing places abnormal conditions on a program in order to analyze a program's ability to handle increased demands. Acceptance testing is conducted by the user under the watchful eye of the developer.

f. System Maintenance/Support. After a system has been tested and accepted by the user, it is considered to be in the maintenance phase. [Ref. 8:pp 20-22] Maintenance consists of fixing errors, adding user desired enhancements and additional functionality, and adapting to hardware and software system changes.

The SDLC also has key personnel that make the process function; these positions are Database Administrator, Systems Analyst/Project Manager, Business Analyst, Programmer, Data Administrator and User.

The Database Administrator (DBA) is the individual responsible for the selection, evaluation, implementation, and management of a database management system. This person is an organization's leading technical expert on database activities

---

[6] Other types of performance tests include: volume, security, configuration, recovery and human factors.

and is responsible for the daily database operations. [Ref. 22:p. 44]

The Systems Analyst/Project Manager is responsible for the overall development of an information system. They design and modify systems by turning user requirements into a set of functional specifications, which are the blueprints of the system. Systems analysts are the architects, as well as the project leaders, of an information system. It is their job to develop solutions to user's problems, determine the technical and operational feasibility of their solutions, as well as estimate the costs to develop and implement them.

The Business Analyst analyzes the operations of a department or functional unit. Their purpose is to develop a general system's solution to the problem. It may or may not require automation. The business analyst provides insights into the business operation for the systems analyst.

The Programmer is proficient in a particular computer language and good software engineering practice and writes application programs based on provided functional specifications. The programmer will conduct unit tests and is normally proactive in the integration testing of a system.

The Data Administrator has the overall responsibility for the organization's data resources, and is responsible for non-technical activities such as planning and defining the conceptual framework for the overall database environment, not

just that specifically limited to DBMS usage. [Ref. 22:pp. 43-44]

The User is the person or organization that will own and operate a software system after it has completed a successful acceptance test. It is their responsibility to determine the systems requirements and functionality.

The SDLC can be enhanced by CASE tools. CASE tools are a collection of hardware and software elements that "aid" the user in software development. The "aid" that both CASE and I-CASE provides is that they automate processes of software development that were previously manual operations. CASE tools can be broadly categorized into three parts: upper CASE, lower CASE and integrated CASE (I-CASE). [Ref. 4:p. 45] Upper CASE tools consist of software tools that aid in the analysis and design phase of the software development life cycle. Lower CASE generally deals with the latter stages of the software development life cycle: code construction, code testing and actual implementation. I-CASE tools combine the separate functionality of both upper and lower CASE tools into a single set of interworking tools. [Ref. 4:p. 45] Further detail of CASE and I-CASE is covered in Chapter III.

7. The Conceptual Re-engineering Model

Now that the re-engineering process, SDLC, and a brief over-view of CASE has been discussed, a model is presented that describes the re-engineering process that relates all three. Figure 2-3 displays a re-engineering cycle developed

by Charles Bachman.[7] This is a model that depicts the role people perform in a re-engineering environment that is utilizing CASE. The model was altered for this paper to also show the corresponding SDLC levels of development. This model is different from the conventional SDLC in that CASE usage enables a "continuity of applications systems and their revisions over time." [Ref. 9:p. 50] In other words, the conventional SDLC without CASE was a cradle to grave development scheme. Using CASE and I-CASE tools provides continual modifications and enhancements through re-engineering.

The model works as follows: reverse engineering starts at the bottom left corner with an existing application at the operational level. CASE or I-CASE tools allow programmers, who are normally more acquainted with the program source code, the ability to extract and refine existing specifications that will be raised to higher levels of abstraction and eventually placed into a repository. The initial design specifications identified will be reviewed by the programmer and the DBA at the implementation level. The implementation level will categorize source level descriptions of files and databases. The design objects in this phase include: records, reports, and screens. The information identified in the implementation level will then be passed on to the data analysts and systems

---

[7] It appeared in the July, 1988 issue of Datamation.

analysts at the specifications level where the data model[8] of the application will be developed. The specifications level will identify the objects of the application, which will be in the form of entities, relationships, procedures, and processes. The requirements level will involve the business analysts identifying the goals, requirements and critical success factors that the application should embody. [Ref. 9:pp. 50-56] The reverse engineering process culminates with the population of the design specifications into a repository.

Once the design specifications are resident in a repository, the forward engineering process may begin. The forward engineering process involves more than utilizing CASE or I-CASE tools. People must determine how to use the information in the repository and integrate that information, using CASE or I-CASE tools, in a coherent manner that will enable the construction of a new system. Testing is not shown in this model, not because it is a transparent or minor process; it is not. But because it is assumed that testing always takes place in a software development process. According to Tom McCabe, of McCabe & Associates, Inc., "testing consumes one half to one third of most software project budgets." [Ref. 46:p. 8]

The model shown in Figure 2-3 may give one the feeling that re-engineering is a simple process when using a CASE or

---

[8] A data model describes how data is structured in a database. Examples of databases include hierarchical, network, and relational.

I-CASE tool. The reality is that re-engineering is a highly complex and difficult task. A prime example is a re-engineering case study conducted by the Internal Revenue Service (IRS) and the National Institute of Standards and Technology (NIST). The case study was focused on the IRS Centralized Scheduling Program (CSP) system. This system was written in 1983 in COBOL 74 and consisted of 37 source programs that constituted approximately 50,000 lines of COBOL code. [Ref. 38:p. 7] Additionally, the CSP system had 53 subroutines of assembly language that totaled 2,738 lines of code (LOC). [Ref. 38:p. 7] The system was not completely re-engineered. Approximately 56% of the system was reverse engineered to the design level and approximately 38% of the system was re-engineered (source code produced). [Ref. 38:p. 11] Even though CASE tools were used in the project it was revealed that:

> Analysis by humans is essential for identifying what information is important, determining the functionality of each program and the entire system, and judging whether the functionality is necessary . . . . Some steps in the re-engineering process seemed cumbersome and time consuming. It was possible to automate some steps, but human effort was needed for analysis and tool operation. [Ref. 38:pp. 5-11]

CASE and I-CASE tools do not relieve the need for human intervention in the software development process. Re-engineering requires people to sometimes manually read source code line by line in order to get a feel of what the original developers of the system were trying accomplish.

B. RE-ENGINEERING POTENTIAL

There are limited published case studies that discuss organizations' experiences and lessons learned with re-engineering. Consequently, at present, there is little empirical evidence to support re-engineering benefits. One must realize, however, that re-engineering is still a young industry. But this does not mean that there is not sufficient evidence to indicate that benefits are to be gained from re-engineering. Re-engineering has strong potential. In order to understand the potential benefits of re-engineering, it is helpful to first view the current state of the software inventory in industry; the motivating factors, or reasons to re-engineer; and the importance software metrics play in evaluating existing systems for re-engineering.

1. Characteristics of Industry Software Inventory

The significant emphasis given to software re-engineering by industry and the DoD is not by accident. The underlying reason for the current emphasis toward re-engineering is no doubt influenced by the current characteristics of the software inventory in industry:

a. There are many old and aging applications. The average software application is greater than ten years old. [Ref. 40]

b. Poorly structured applications account for approximately 75% of the existing industry total. [Ref. 40]

c. Undocumented applications account for approximately 35% of the industry total. [Ref. 40]

d. Over 50% of industry personnel are being utilized software maintenance. [Ref. 40]

Given the above software inventory posture, it is not surprising to witness consulting firms positioning themselves as re-engineering contractors. The combined market for re-engineering services and products is estimated to reach $14 billion by 1995. [Ref. 11:p. 51]

2. Reasons to Re-engineer

In addition to the above listed characteristics of the software in industry, re-engineering has gained prominence for other reasons. First, re-engineering technology provides the means for an organization to extend the life of a system. Making program code more maintainable at the source code level, in turn enables programmers to capture important elements of the original system at the design level. Better systems will result with fewer flaws. [Ref. 3:p. 32] Also identifying significant business rules at the design level, allows them to be loaded into a repository for future reuse. [Ref. 3:p. 32]

A second reason for re-engineering is that it can save money if executed properly. The cost of software maintenance is a huge expenditure of present information technology budgets. The worldwide estimate of software is over 100 billion lines of code, with COBOL comprising 80% of the total. [Ref. 3:p. 32] Further-more, maintenance of existing applications are estimated to consume more than 70% of

programming resources. [Ref. 12:p. D12-2] Proper re-engineered systems produce more maintainable software. Re-engineering existing, especially older, software systems provides a new aspect to the "make or buy" decision that many organizations face. It also offers to reduce the software maintenance financial burden by making code more maintainable. Perhaps the greatest reason organizations are considering re-engineering alternatives is that many older systems are still quite useful. But they must be able to be modified rapidly to respond to changing business conditions and strategies. [Ref. 54:p. 1] As written today, they cannot. Re-engineering these often unstructured, undocumented older systems positions them for rapid modernization.

Today, the DoD finds many of its software systems difficult and costly to maintain. Many are unstructured and documentation has often deteriorated and is quite poor. In research conducted by Software Productivity Research, Inc., when compared to other large industries such as insurance, banking, manufacturing, telecommunications and oil, the DoD allocates approximately 70% of its programmers to maintenance. [Ref. 40:p. L-11] This high percentage devoted to maintenance is not surprising. The same research indicated the DoD leads all other industries in the following:

a. DoD portfolio size: portfolios total 300,000,000 statements and 1,750,000 function points.[9] [Ref. 40]

b. DoD portfolio average age: 13 years. The U.S. portfolio age is 10 years. [Ref. 40]

3. Metrics

Before an organization can determine its strategy for re-engineering, it must first assess its current state of data and processing capability. Using software metric tools to evaluate the vitality of existing programs is usually the first step in the re-engineering process. Two common metric measurements in use today are function point analysis and cyclomatic complexity analysis.

The first, function point analysis, is a function of the weights of five different factors: inputs, outputs, inquiries, interfaces and logical files. [Ref. 7:p. 46] Function point analysis is language independent and provides not only a measure of functionality and early complexity estimation, but also serves as a indicator of program code quality and software organization productivity. "One of the advantages of the Function Point metric is that it can be used to predict and measure all sources of software errors, and not just coding errors." [Ref. 44]

---

[9] A portfolio is defined as a set of active programs and systems owned by an enterprise (typically 1 to 50 systems and 10 to 10,000 programs). Portfolio sizes range from 100,000 to 100,000,000 source statements. [Ref. 40:p. L-8] Function points are discussed in greater detail in the next section of this chapter.

Cyclomatic complexity tools measure and assess the branching logic of a program module. The greater the number of unique and distinct paths through a module's source code, the larger the value of the cyclomatic complexity. Empirical evidence has revealed that modules with less than five paths through its program logic will be easy to understand; between five and 10 paths is considered not too difficult; 20 and greater paths is considered high; and when the number of paths exceed 50, the software is considered untestable. [Ref. 7:p. 237]

Tables 2-1 and 2-2 in Appendix D, display data collected from more than 4000 software projects studied by Software Productivity Research, Inc.[10] Table 2-1 shows that as the size of a system increases in terms of lines of code (LOC), the number of enhancements also increases. LOC is simply the number of lines contained in the source code of an application. An enhancement is defined as a new function added to an existing system to meet new requirements. [Ref. 40:p. L-7] Table 2-1 also indicates that as system size and enhancements increase, the productivity rate of programmers decreases in both LOC per person year and function points per

---

[10] Both tables were presented by Capers Jones, Chairman of Software Productivity Research, at the National Software Re-engineering & Maintenance conference on August 10, 1992. The data contained in both tables appear in different forms in Mr. Jones' book Applied Software Measurement (McGraw-Hill, 1991). Mr. Jones notes that his studies have a high potential for error content. This is attributed to the fact that, according to Mr. Jones, "there are no current U.S. or international standards for consistent counting of software tasks and deliverables." [Ref. 7:p. 124]

person year. This data indicates that as systems become larger programmers are less productive . This is in a large part influenced from the point that as programs get larger they generally become more complex.

Table 2-2 depicts the impact of poorly structured programs verses well structured programs. In the COBOL programming language, poor structure is characterized by a program that has numerous branches through its logic, e.g., many GOTO statements. The first attribute in Table 2-2, Defect Potential, accounts for defects from five different origins: requirements, design, coding, documentation, and bad fixes. [Ref. 44] Bad fixes are simply corrections to errors, which in actuality create more errors. The next attribute is Removal Efficiency, which is the percentage of errors removed from a system before delivery to the user. In Table 2-2, the removal efficiency of the poorly structured system is 10% less than that of the well structured program. This is a substantial difference. A program with a low Removal Efficiency will have more errors surface later after a program has been delivered to the user. This can be costly in terms of maintenance. The U.S. average for removal efficiency is 85%, which is not a stellar percentage. [Ref. 44] The ultimate goal is to have 100% removal efficiency. Large corporations such as Motorola, Raytheon, Hewlett-Packard, and IBM have achieved removal efficiency levels of 99% [Ref. 44] Stabilization Period is the time it takes for a program, after

it has been delivered to the user, to have errors debugged and start working as specified. Mean Time to Failure is simply the average time between failures in a program. There are two fundamental principles that may be deduced from these tables:

a. As a system grows in size, programmer productivity decreases. Therefore, an organization should know the extent and nature of its data inventory before attempting a re-engineering project. An organization must know "where it is," in terms of their data, before it determine "where it wants to go" if re-engineering is under consideration.

b. If metric analysis is not conducted, an organization will not have an accurate picture of its applications. An IBM study on maintenance costs revealed that if a system required more than 12% of its code to be changed, it would be prudent, economically, to scrap the system and start over. [Ref. 20:p. 4] Metric analysis will aid an organization in understanding trouble spots in its systems and thus provide a means for evaluating the need to re-engineer. Re-engineering will provide better structured code. Otherwise, an organization may continue to maintain costly systems.

Metrics analysis cannot be overlooked in a re-engineering effort, especially for large systems. It should be one of the first areas of consideration when re-engineering. Function point and complexity analysis are two means used to diagnose the problem areas of a program. Metric analysis allows developers of a system the ability to understand the "vital signs" of a program. That is to say, trouble spots in a program's logic can be isolated and corrected. This will in turn make programs more structured and maintainable.

4. Savings

Software re-engineering is a growing market and leaders in the software field champion its use for the right programs, and its potential benefits. However, there presently is little empirical or quantitative evidence to support that software re-engineering will always provide cost savings. In fact, there have been some costly failures. Not every old program should be re-engineered. However, a significant amount of anecdotal data does support that software re-engineering can provide significant savings over the maintenance life of a system.

Re-engineering is not a generic process. That is to say, each application considered for re-engineering is different, just as each organization's corporate culture is different. Thus, a re-engineering process that worked for one organization may not necessarily work for another. But, with good software engineering program management and employment of a disciplined methodology, the risk of failure may be significantly reduced.

One example of cost savings from a software re-engineering effort was an Army project. The Army Institute for Research Management Information, Communications and Computer Sciences (AIRMICS), the U.S. Army Information Systems Software Development Center-Atlanta, and the Software Engineering Research Center of the Georgia Institute of Technology completed a re-engineering project using CASE

tools. The re-engineered Army Installation Material Condition Status Reporting System (IMCSRS) was completed in June 1991 and was distributed to 40 Army installations. The total cost for re-engineering the system was $186,394. [Ref. 45:p. 26] The estimated net present value cost-benefit for all the sites using the system, based on an estimated 10 year lifetime for the system, was estimated to be $3,187,240 after deducting the re-engineering development costs. [Ref. 45:p. 26] The net present value analysis revealed the following [Ref. 45:p. 26]:

a. internal rate of return (IRR): 131.4%

b. recovery of development costs expected within 0.44 years

c. overall benefit to cost ratio:

Even though the Army re-engineering effort is an isolated case, it shows growing evidence that re-engineering can provide significant rewards.

5. Potential Benefits

The main objectives of most commercial businesses are to increase market share and produce a profit. When technology offers not only a means to increase profit, but also a means to maintain and build better products with enhanced capabilities, organizations will most likely devote a portion of their resources to acquire such technology. Software re-engineering offers these potential benefits.

Even though re-engineering is still in its infancy, it is receiving significant attention from government and private

industry. The following is a list of five re-engineering benefits [Ref. 3:pp. 33, 36]:

a. Reducing Software Maintenance Efforts. Maintenance consumes a large portion of information technology budgets. Re-engineering can produce better structured code, which is easier to maintain, and thus less costly.

b. Preserving Investment. The cost of software is high. Not only the cost of the software itself, but the cost of the manpower needed to write, manage, and maintain the software is substantial. Organizations naturally want to maximize their investment in software to ensure that it continues to function productively.

c. Increasing the Productivity of System Maintainers. CASE and I-CASE tools automate many tasks that were once manual. As developers become more proficient in using CASE and I-CASE tools in a re-engineering environment, they will increase their skill level. This will provide additional time for developers to concentrate on more productive tasks.

d. Enabling System Conversion and Migration to New Hardware. As vendors upgrade their hardware, some organizations may wish to change (migrate) to new computing platforms.

e. Enabling the Reusability of Existing System Components and protecting and extending the system's life. Repositories facilitate this capability.

Interestingly, as a result of software re-engineering, the total lines of code of a new system may increase or decrease. This is usually dependent upon the original system's size, complexity, and programming language. The key benefit of a successful software re-engineering effort is that a new system will be better structured. It will also be better documented. This will allow for easier future

maintenance. The following section will discuss some of the limitations associated with software re-engineering.

C. RE-ENGINEERING LIMITATIONS

While software re-engineering offers definite benefits, it is not a panacea to answer all the problems involved with software maintenance. For instance, re-engineering using CASE and I-CASE tools is not without its challenges. One of the major problems facing the CASE and I-CASE industry is that most of the tools, specifically data repositories, are proprietary. Many systems thus cannot communicate and work with each other and organizations are often limited in their selection of re-engineering tools. It should be noted that re-engineering tools are no substitute for good management, methodology, or software engineering techniques. Re-engineering limitations are both technical and managerial.

1. Technical Considerations

Since re-engineering encompasses reverse engineering, existing program code must first be brought to a higher level of abstraction in order to be forward engineered to a new form. A problem may arise if the higher abstraction level does not capture all the significant properties of the original program code. There is presently a fundamental limitation of re-engineering technology in recognizing the difference between semantic (logical design) and syntactic (physical design). Technology is at a state where CASE tools can understand the syntax of source code, e.g., a branching

operation or a loop. But these tools do not understand the semantics, i.e., what is actually meant in the source code. This is a primary reason why re-engineering requires domain experts to physically intervene in the process. [Ref. 39:p. 5]

According to Ravi Koka[11], the missing link in reverse engineering is the loss of the original business intentions of a system between the physical model (source code) and logical model (higher abstraction). [Ref. 12:p. D12-2] According to Mr. Koka, the major factors that contribute to the "missing link" are [Ref. 12]:

a. Cryptic Source Code. Numerous older systems were developed more along artistic rather than structured guidelines. As a result these programs are hard to understand. The only people that truly know the system are the people who created it.

b. Personnel Turnover. All corporate knowledge of a program may reside in the people who created it. When they leave the organization, the system, if left in a cryptic form, will be hard to maintain.

c. Patches to Programs. They tend to make programs less structured and understandable.

d. Poor Documentation. This is in the form of both poor source code documentation as well as poor and outdated system's manuals. Without adequate documentation subsequent programmers and analysts encounter great difficulty in determining the business rules and structure of a system. In some cases the documentation is so poor the system cannot be re-engineered and must be created from scratch.

---

[11] Mr. Koka is President of Software Engineering and Enhancement Center, Inc. Mr. Koka was a guest lecturer at the CASE World Conference & Exposition in Santa Clara, California on February 20, 1992.

The current state of re-engineering technology only allows source code to be reverse engineered to the design level, not to the analysis level. [Ref. 10:p. 26] This is presently a re-engineering limitation.

2. Management Considerations

Some simple facts about software re-engineering are that it is expensive, time consuming and complex. A typical three to six week reverse engineering training session by a major consulting firm can range from $50,000 to $400,000 for a large system. When other services, such as forward engineering utilizing CASE tools are added, the cost can range from $200,000 to $1 million for the initial re-engineering project. [Ref. 11:p. 52] Depending on the condition of the existing code, the reverse engineering cost can range between $30,000 to $100,000. Resystemization entails migrating an existing system to a new environment through the use of CASE tools. The cost for resystemization can range between $100,000 and $600,000. The cost for the re-engineering services can range between $500,000 to $1,000,000.[12]

It is essential that organizations first conduct a thorough analysis of their data inventory and assess their requirements for re-engineering. Management must insist upon it. Such a self assessment is critical because re-engineering may not always be in the best interest of an organization.

---

[12] This information was collected in a phone conversation the author had with Richard Phelps of Ernst & Young on May 18, 1992.

For instance, if a system's documentation is poor, or non-existent, re-engineering can't reincarnate it into a new system. Thus, an organization would be better off starting from scratch to develop a new system. One consequence of not conducting an internal analysis may be that after an organization expends the capital for CASE, or I-CASE tools, the tools end up becoming shelfware and not used because the system cannot be successfully re-engineered. The old adage "you can't fix what you don't understand," is quite applicable to software re-engineering.

Management should consider a methodology that brings organization and discipline to the re-engineering process. The following is one high level approach that consolidates a re-engineering project into five major steps [Ref. 39]:[13]

a. Determine the systems functional requirements: what should the new system accomplish?

b. Make a technical assessment of the current System: how does the current system accomplish its functions?

c. Develop a new conceptual system model: how should the new system complete its functions, i.e., what hardware/software configuration is needed?

d. Develop a scenario analysis: what are the alternatives in implementing the new system, i.e., does the organization need to re-engineer? How do the alternatives compare in terms of percentage of code reuse, tools used, cost savings and risk?

---

[13] This approach was presented by James Rothe of Andersen Consulting at the National Software Re-engineering & Maintenance conference on August 12, 1992, in San Jose, California.

e. Develop an implementation plan: this includes detailed work-plans, tool assessment and selection, project phasing and risk management.

Many older systems, some being mission critical, are reaching the end of their useful life. Organizations are faced with the task of updating these systems to conform to present demands. But with shrinking budgets and fewer trained people, this is getting harder. Re-engineering falls within the strategic and operational levels of corporate decision making. Re-engineering requires talented people with adequate training and senior management endorsement. But senior management endorsement encompasses more than vague knowledge of software technology. Senior managers must understand the full impact that re-engineering will have on the culture of the organization, and the consequences and pitfalls that may be encountered when attempting a re-engineering project. [Ref. 13:p. D24-11] Some common re-engineering pitfalls are listed in Table 2-3. Everyone associated with re-engineering, from senior management to the software participants, should be aware of such pitfalls and take action to mitigate the impact of each one.

D. A TAXONOMY OF A RE-ENGINEERING PROJECT

Re-engineering may not only consume a large portion of an information technology budget, but also requires integration with corporate level strategic and tactical planning. It is by thorough planning that re-engineering objectives are

defined and disseminated throughout an organization. The following sections deal with the criteria for selecting a project and factors that contribute to a successful re-engineering effort.

1. Re-engineering Selection Criterion

Not all systems are candidates for re-engineering. The following criterion offer a guideline to use for assessment of target systems for re-engineering [Ref. 2]:

a. Importance of the program or system to the company's operation.

b. Ease of maintenance: program metric tools allow personnel to measure the complexity of a system's source code and determine the quality of the code and ease of maintenance. Generally, programs with a high level of complexity are good candidates for re-engineering.

c. Current reliability: this is a measure of failure rate within a system. If a system has numerous revisions to its code, chances are the system will have a high failure rate. Systems with high failure rates are candidates for re-engineering.

d. Frequency of maintenance. If a system is frequently requiring maintenance it is likely to become unstable. Programs like this are prime candidates for re-engineering.

e. Timing. If a program is maintained by the creator of the program or a maintenance programmer who has intimate knowledge with the program, then the program may not be a good candidate for re-engineering. This is because of programmers' proprietary attitudes toward their creations. They are reluctant to re-engineer their programs. Only through personnel turnover and the addition of new personnel to the maintenance of a system will proprietary attitudes relax, thereby easing the resistance to re-engineering.

2. Cost Factors

There are models such as the constructive cost model (COCOMO) that provide in-depth analysis for judging the costs in developing a software system. However, there are no formal cost models used exclusively for re-engineering.

> . . . it is important to note that even traditional estimation models are not wholly applicable to re-engineering projects. For example, re-engineering project development costs are reduced since test cases and design information are already completely or partially present (left over from the original development process). [Ref. 30:p. 16]

Costs incurred during software re-engineering will vary among different organizations. However, there are some common variables that organizations should consider. A re-engineering study conducted by the National Institute of Standards and Technology (NIST) and the Internal Revenue Service (IRS) concluded:

> The cost-effectiveness and feasibility for re-engineering a particular software system will be dependent on a number of variables that are specific to that system and the approach taken. These variables are: the goals for re-engineering, condition of current application system and documentation, tool(s) support, and involvement of knowledgeable personnel. [Ref. 38:p. 13]

Appendix A displays a prototype methodology for determining if re-engineering is a feasible option for an organization to take. The methodology was developed by the Software Technology Support Center at Hill Air Force Base, Utah. [Ref. 30]

3. Steps Involved With a Successful Re-engineering Process

In September 1990 Price Waterhouse opened its Re-engineering Center in Tampa, Florida to assist clients with their re-engineering efforts. According to Steve Errico, a partner at Price Waterhouse, there are nine distinct steps involved with a successful re-engineering process. Each step in the re-engineering process requires the use of CASE tools. [Ref. 54:p. 3] The nine steps are as follows [Ref. 54]:

a. Metrics: analysis of code quality and complexity;

b. Inventory: identification and location of affected components, i.e., this identifies parts of a program that re-engineering will have an impact on;

c. Analysis and documentation: understanding functional characteristics of the existing system, this includes identifying program components that may require engineering;

d. Data reverse engineering: understanding the nature of existing data, i.e., identifying the relationships and meaning among the data and obtaining control of the data;

e. Process reverse engineering: understanding the nature of existing program code, i.e., understanding the semantics embodied in the code;

f. Data conversion: moving data into the new database environment, i.e., moving from a hierarchical to a relational database;

g. Analysis and design of the new system;

h. Code generation;

i. Testing and verification.

E. SUMMARY

This chapter has provided a broad overview of software re-engineering. Re-engineering is a function of reverse engineering and forward engineering. Definitions were provided to lay the ground work for understanding the re-engineering process, followed by discussion of the relationship and distinction between re-engineering and reverse engineering. Reverse engineering is the first step in the re-engineering process. It consists of using CASE or I-CASE tools that enable developers to raise program source code to higher levels of abstraction. Reverse engineering culminates with the population of design elements into a data repository. Forward engineering is the second part of re-engineering. It takes the design elements within a data repository and uses CASE and I-CASE tools to create a newer version of the original system. But CASE and I-CASE tools are only aids in the re-engineering process. Successful re-engineering demands skilled and trained people, both managers and technical personnel, to become proactive in the re-engineering process.

Re-engineering embodies both benefits and limitations. In order to exploit potential benefits, it is necessary to understand the nature of a system's source code. This chapter discussed the importance of metric analysis as a means to identify as well as understand program source code. Re-engineering is not without limitations. This chapter covered

both technical and managerial considerations that focus on re-engineering limitations.

The chapter concluded with a look at how applications are considered for re-engineering, followed by suggested steps for a successful re-engineering effort. However, there are few documented case studies that provide empirical and quantitative evidence for software re-engineering. This is attributed to the fact that re-engineering is still a young and emerging industry.

III. INTEGRATED-COMPUTER AIDED SOFTWARE ENGINEERING

The complexity of software development and the requirements placed on organizations to meet customer needs in a rapidly changing technology environment, have placed software developers in a position where creating, maintaining or re-engineering a system must be accomplished efficiently and in a reasonable amount of time. Maintenance costs and problems with data management plague both government and private industry.

The evolution of CASE tools has enabled the automation of certain parts of the software development cycle and has eased some of the difficulties with data management and maintenance. There are individual CASE tools that can address particular areas of software development. For example, Knowledgeware has three CASE products: Inspector, Pinpoint, and Recorder, which do this. Inspector is a tool that measures the quality and complexity of COBOL applications. Pinpoint is a maintenance tool that enables programmers to see the interworkings of a program, whether the program is unstructured or not. Recorder is a restructuring tool. It can remove inexecutable program logic and replace it with better structured program syntax and reduce the number of test paths, thereby making updated code easer to read and understand. [Ref. 47:pp. 5-9]

This chapter starts with a definition for software tool integration, followed by the difference between CASE and I-CASE tools. Next, the key elements that constitute I-CASE tools is presented. The chapter ends by discussing the methodology, benefits and limitations of I-CASE and a synopsis of the DoD I-CASE procurement.

A. A DEFINITION OF INTEGRATION

What is meant by tool integration? In simplistic terms, integration means "components function as part of a single, consistent, coherent whole." [Ref. 32:p. 30] But to say that CASE tool A is well integrated with CASE tool B requires clarification. This is because both CASE tool A and CASE tool B have similar and different characteristics. The following sections review the four types of integration: presentation, data, control, and process.[14]

1. Presentation Integration

This form of integration allows users to interact with different tools in the same way. [Ref.34:p. 8] The goal of presentation integration is to improve the efficiency and effectiveness of the user's interaction with the environment by reducing his cognitive load. [Ref. 32:p. 30] In other words, presentation integration enhances productivity by alleviating the need for the user to learn a different way to

---

[14] In the March, 1992 issue of IEEE Software, Ian Thomas and Brian Nejmeh propose a framework, based on previous work by Anthony Wasserman, which defines integration and identifies the goals of integration.

interact with each tool. [Ref. 34:p. 8] A simple analogy of this is a pull down menu screen. These are menus that basically spoon feed a user in moving from one tool to another by displaying similar alternatives via screen display. This allows the user the ability to initiate or terminate a task. By keeping the menu screens simple and intuitive, a user does not spend excessive time learning the characteristics of a new tool.

2. Data Integration

The goal of data integration is to ensure that all the information in the environment is managed as a consistent whole, regardless of how parts of it are operated on and transformed. [Ref. 32:p. 30] CASE tools are considered well integrated when they share a common view of data. [Ref. 32:p. 32] For example, IBM's AD/Cycle Information Model. The AD/Cycle Information Model defines the format and structure of information stored in the repository. [Ref. 19:p. 25] The definitions stored in the repository are understood by the different tools, which enables consistency, or a common view of data. [Ref. 19:p. 25]

3. Control Integration

The goal of control integration is to allow the communication and sharing of information between CASE tools. [Ref. 32:p. 30] Control integration provides a transparent means for users to communicate between tools. The user does not need to know the interworking mechanisms of each tool that

is used. [Ref. 34:p. 8] For example, when a user clicks a mouse, or hits a keystroke, they do not need to know the electrical engineering aspects of the circuit gates that the binary information transits.

4. Process Integration

The goal of process integration is to ensure that tools interact effectively in support of a defined process. [Ref. 32:p. 30] In other words, the concept behind process integration is the ability for several tools to work in concert from analysis and design to code construction of a system. A good example of this is Texas Instruments I-CASE product Integrated Engineering Facility (IEF). In a Computer-world survey of 143 organizations using I-CASE tools, IEF received the highest rating in integration as well as the highest ratings overall. [Ref. 48:p. 72]

B. INTEGRATION SUBCOMPONENTS

The four types of integration mentioned above are further broken down into subcomponents that help explain how each particular type of integration works. Figure 3-1 displays some of the properties that compose each integration type and the interaction they have upon a single CASE tool. Each integration property shown in Figure 3-1 is discussed below.

1. Appearance and Behavior

This property addresses the ease the user has in interacting with a tool, having already learned to interact with another tool, i.e., "how similar are the tools' screen

appearance and interaction behavior." [Ref. 32:p. 31] Two tools are considered well integrated with respect to appearance and behavior if a user's experience with and expectations of one can be applied to the other." [Ref. 32:p.

2. Interoperability

This property addresses the issue of two tools being able to view data as a consistent whole. [Ref. 32:p. 32] Two tools are considered well integrated with respect to interoperability if "they require little work for them to be able to use each other's data." [Ref. 32:p. 32]

3. Nonredundancy

This property addresses and identifies redundancy of data between two tools. An example of redundant data would be several names for social security in a database, like SSN, SOC_NUM, or SNUM. Integrated tools should minimize redundant data. [Ref. 32:p. 32] Two tools are considered well integrated with respect to nonredundancy if "they have little duplicate data or data that can be automatically derived from the other data." [Ref. 32:p. 32]

4. Data Consistency

This property addresses the issue of tools being able to manipulate data and pass the data on without loosing the meaning of the data. Two tools are considered well integrated with respect to data consistency if:

> . . . each tool indicates its actions and the effects on its data that are the subject of semantic constraints that also refer to data managed by another tool. [Ref. 32:p. 32]

5. Data Exchange

In order for two tools to exchange data, the tools must agree on data format and semantics. [Ref. 32:p. 32] This property addresses the issue of data generated and sent by one tool and the ability of a second tool to manipulate the data sent to it. [Ref. 32:p. 32] Two tools are considered well integrated in respect to data exchange if "little work on format and semantics is required for them to be able to exchange data." [Ref. 32:p. 33]

6. Provision

A tool is considered well integrated in terms of provision integration if "it offers services other tools in the environment require and use." [Ref. 32:p. 33] For example, a project management tool requires textual task descriptions. But in order for the text to be entered, it relies on the services offered by the editing tool. [Ref. 32:p. 33]

7. Process

A process step is the decomposition of a task performed by different tools to carry out a process. [Ref. 32:p. 34] In other words, to carry out a task, executions performed by different tools achieve the accomplishment of a task. In order for this to occur, any single tool's preconditions must be met. "A tool's preconditions are satisfied when other tools achieve their goals." [Ref. 32:p. 34] Tools are

considered well integrated in terms of process step integration if:

> . . . the goals they achieve are part of a coherent decomposition of the process step and if accomplishing these goals lets others achieve their own goals. [Ref. 32:p. 34]

8. Event

There are two parts to event integration. First, a tool's preconditions should reflect events generated by another tool. Second, a tool should generate events that aid in satisfying other tools' preconditions. [Ref. 32:p. 34] Tools are considered well integrated in terms of event integration

> . . .they generate and handle event notifications consistently (when one tool indicates an event has occurred, another tool responds to the event). [Ref. 32:p. 34]

9. Constraint

Each tool in a CASE environment has constraints by which it operates. In other words, a tool may be designed to perform functions within specified limits. Constraint integration is described as:

> There are two aspects to enforcing a constraint. First, one tool's permitted functions may be constrained by another's functions. Second, a tool's functions may constrain another tool's permitted functions. Tools are said to be well integrated with respect to constraint integration if they make similar assumptions about the range of constraints they recognize and respect. [Ref. 32:pp. 34 - 35]

The framework discussed in this section attempts to define integration in four areas: process integration, data integration, control integration, and presentation integration. Each of these respective types of integration are further decomposed to detailed attributes that explain their specific type of integration. This framework is by no means the definitive answer to completely explain integration. However, it provides the novice an intuitive explanation of what is involved with the concept of integrating CASE tools into a cohesive environment.

C. CASE VERSUS I-CASE

In contrasting CASE to the systems development life cycle (SDLC), CASE breaks out into upper CASE and lower CASE. Upper CASE deals with the overall planning environment that a system must operate in, equivalent to the analysis and design phases of the SDLC, where the logical model of a system is defined. Additionally, upper CASE tools create data flow diagrams and entity-relationship diagrams that aid in the development of the logical model. The major event in the upper CASE environment is development of a data dictionary and its population with elements that will define the system. Lower CASE facilitates the actual code construction, testing and implementation of a system. [Ref. 16:p. 32]

In comparing I-CASE and CASE, there are two major distinctions that make I-CASE unique from CASE. First, CASE tools only work on specific parts of the development life

cycle. Different third party tools can not currently be brought together and function as an integrated tool set. [Ref. 33:p. 19] I-CASE tools are still proprietary products, but they can integrate all aspects of the development life cycle under one set of single vendor tools. Second, I-CASE tools generate 100% executable program code directly from the design specifications and models created earlier in the development process and stored in a centralized repository. [Ref. 14:p. 6] Some CASE tools can also generate code, but only partial code for screens, reports and data definitions. [Ref. 27:p. 45] However, other CASE tools can generate compilable and executable code. So what is the distinction? The distinction lies in that within an I-CASE environment, all the tools share and understand the data. For example, if you change an attribute to a data name, the change is automatically made so that the other tools understand the change. The programmer does not need to manually go in and update the change with each tool. CASE tools cannot do this. To produce compilable and executable code with CASE tools it takes manual human intervention with each tool to ensure data consistency.

D. I-CASE REPOSITORY

The repository is the heart of the I-CASE system. A repository is defined as a database that serves as the mechanism for storing and organizing all information concerning a software system; it is the single place in which data can be entered once, kept consistent, and made available when

needed. [Ref. 23:pp. 53, 57] The key aspect of the repository is that it not only stores relationships among data, but stores the meaning of the data as well. This is often referred to as meta-data.[15] According to James Martin, there are two types of repository used in the CASE environment, a dictionary and an encyclopedia. The definition and distinction for both are as follows:

> **ENCYCLOPEDIA.** A repository of knowledge about the enterprise, its goals, entities, records, organizational units, functions, processes, procedures, and application and information systems . . . . A **dictionary** contains names and descriptions of data items, processes, variables, etc. An encyclopedia contains complete coded representations of plans, models and designs with tools for cross checking, correction analysis, and validation. Graphic representations are derived from the encyclopedia and are used to update it. The encyclopedia contains many rules relating to the knowledge it stores, and employs rule processing, the artificial intelligence technique, to help achieve accuracy, integrity, and completeness of the plans models, and designs. The encyclopedia is thus a knowledge base which not only stores development information but helps to control its accuracy and validity. The encyclopedia should be designed to drive a code generator. The toolset helps the systems analyst build up in the encyclopedia the information necessary for code generation. The encyclopedia "understands" the modules and designs; a dictionary does not. [Ref. 28:p. 461]

It is important to point out that a **knowledge base** simply contains the rules of a system; it does not perform any artificial intelligence actions by itself. It is the **rule processing** technique within the knowledge base that allows data to be defined and formed as objects and allow the objects

---

[15] Meta-data or "data about data" defines how data is structured in a repository, e.g., as a record, business entity, or business process.

to form relationships that can be further shared by the system. For example, rule processing determines how processes on a dataflow diagram, or elements of an entity-relationship diagram, are linked and referred to. [Ref. 15:p. 17]

Within every organization data can be classified in two areas: **business entities** and **business processes**. "Invoice" and "customer" are typical examples of a business entity. Activities performed on a business entity, for example, validation of a customer account number, is considered a business process. During system design and development.

> . . . business entities and processes are identified and documented so appropriate representations of them may be incorporated in procedures, programs, files, and databases. Both the abstract representations and the computer systems are often referred to as data and process models.[16] [Ref. 35:p. 3-6]

The design of a dictionary or a repository begins with the identification of entities and processes. [Ref. 35:p. 3-6]

Meta-data, which is composed of meta-entities, constitutes the basic building blocks of the repository. [Ref. 35:p. 3-7] Figure 3-2 describes meta-data as follows:

---

[16] A data model is a representation or view of collected data. Many current data models use the entity relationship approach. This approach organizes data in terms of entities, relationships, and attributes. A relationship connects two different entities. For example, "officer assigned to engineering department" is a relationship type. An attribute is a data item that describes an entity or relationship. For example, an employee can have a social security number and a wage. A process model shows the flow of data, e.g., dataflow diagrams and entity-relationship diagrams. [Ref. 35:pp. 4-4, 4-9]

> Customer and order are examples of business entities--things of interest to the users of the business systems. "Customer places order" is an example of a relationship among business entities. [Ref. 35:p. 3-7]
>
> Frequently, specific record types in a database contain data about specific business entity types. In this example, there are records containing data about customers, and records containing data about orders. [Ref. 35:p.
>
> Programmers and analysts are interested in the descriptions of the records and databases. Thus, record and database are the entities of interest. Data dictionaries and repositories, designed to store the descriptions of these data entities, contain meta-data about data entities. In this example, there are meta-data records that contain meta-data about records and relationships among records. [Ref. 35:p. 3-7]
>
> Designers and users of data dictionaries and repositories are interested in the descriptions of the various types of meta-data records used to store this meta-data. [Ref. 35:p. 3-7]

The meta-data approach displayed in Figure 3-2 enables a repository to have the flexibility to add more meta-data, and to integrate other software products to the repository.

Not all repositories are structured as knowledge bases. Some repositories such as Digital Equipment Corporation's Cohesion CDD/Repository, uses an object oriented database.[17] Other repositories are developed as hierarchical, network, and relational databases. International Business Machines (IBM) Repository Manager/MVS, structure their repository on data

---

[17] Object orientation view data as separate from the way it is used. In the object oriented approach, data and the procedures that use the data are combined into objects that are described in terms of data and procedures taken together. Anything of interest to the system or the users of the system may be considered an object. [Ref. 35:p. 3-10]

semantics incorporated within an entity-relationship model. Data semantics emphasize identifying data entities as places, persons, events or concepts and defining the relationships and associations between them. [Ref. 19:p. 121]

The repository not only acts as a central common database for data storage, but also allows the sharing of data between diagramming tools. James Martin refers to automated diagram tools like data flow diagrams, action diagrams and entity-relationship diagrams as the means of translating data into different abstraction forms. They also serve as a means for populating the repository with the requisite information needed for planning, analysis, design, construction and maintenance of a system. In order for the diagram tools to achieve this, they must be tightly integrated.[18] Therefore, a rigorous methodology or standard must be enforced to enable diagram tools to share and move data from one representation form to another. [Ref. 14:pp. 6-18]

The **integration standard** is the key to making a repository work. [Ref. 17:p. 4] It assures consistency and quality of data are achieved within a repository. As shown in Figure 3-3 an integration standard is a high level syntax language

---

[18] For example, assume an entity-relationship diagram contains the entity named employee. Employee is a field that contains 10 characters. But during the development process, it was determined that employee needed to have 15 characters vice 10. Tools that are tightly integrated will allow changes to an entity in one phase of development to be automatically updated and carried over into other tools without manual intervention. Furthermore, the meaning of the data is held constant and not altered.

incorporated within a repository. It enables data created and processed by one tool to be shared among different tools without loosing the meaning of the data. [Ref. 17:p. 4] Within an I-CASE environment, all the tools are tightly integrated. According to James Martin, it is the tight integration among tools and the repository that drives the code generator and allows the automatic generation of source code. [Ref. 14:p. 23]

The fundamental requirement to achieving a viable and productive development environment is the repository. What makes the repository unique is the fact that it places the decision maker closer to the system's requirements without the intervention of application programmers. [Ref. 30:p.

E. METHODOLOGY

The use of I-CASE tools themselves do not ensure productivity, quality or success in an application development. The very nature of the demands placed on system development teams, and the requirement for maintainable software, have required development procedures to move from ad hoc "artistic" methods to that of formal and structured procedures embodied by an engineering discipline.

A **methodology** is a set of rules, steps and procedures that are applied to a system to achieve a desired result. Two common methodologies used with I-CASE tools are information engineering and rapid application development. The following two sections will discuss both methodologies.

1. Information Engineering

Information engineering is a methodology that functions as a guideline for project management and development coordination throughout the development life cycle. [Ref. 18:p. 11] It is defined as:

> . . . an interlocking set of automated techniques in which enterprise models, data models and process models are built up in a comprehensive knowledge-base and are used to create and maintain data-processing systems. [Ref. 14:p. 46]

It also spans the entire life cycle of a system including maintenance.

Information engineering consists of four stages: Information Strategy Planning, Business Area Analysis, System Design, and Construction. The process starts with a broad concept of objectives at the Information Strategy Planning phase and successively moves down the remaining three phases. It gains refinement and detail until enough information is collected to implement a system. [Ref. 14:p. 48] The four phases of information engineering consider the following:

a. **Information Strategy Planning:** Concerned with top management goals and critical success factors, a high speed overview of the enterprise, its functions, data, and information needs. [Ref. 14:p. 49]

b. **Business Area Analysis:** Concerned with what processes are needed to run a selected business area, how these processes interrelate, and what data are needed. [Ref. 14:p. 49]

c. **System Design:** Concerned with how selected processes in the business are implemented in proce-

dures, and how these procedures work. Direct end user involvement is needed in the design of procedures. [Ref. 14:p.

d. Construction: Implementation of the procedures using, where practical, fourth-generation languages, code generation, and end user tools. [Ref. 14:p. 49]

Some I-CASE products like Texas Instruments Integrated Engineering Facility (IEF), incorporates the information engineering methodology. There are non I-CASE products like the Ernst & Young Navigator Systems Series, which are also centered around the information engineering methodology. It uses CASE, estimating tools and project management techniques to develop systems. However, it is not tied to any specific CASE tool.

The methodology used by an organization will depend on several factors. These include such things as corporate goals, target application, staff requirements and personnel training levels. Not all I-CASE tools are tied to a specific methodology.

2. Rapid Application Development

Rapid Application Development (RAD) is another methodology used with I-CASE tools. RAD is a departure from the traditional waterfall development methodology model. The waterfall model starts with a feasibility study in which all the requirements of a system are derived. Once the requirements are collected the process of design, programming, testing, integration, and eventual deployment follow.

However, there is a critical flaw with the waterfall model. It is the assumption that all the users needs are captured and identified in the requirements phase. This is seldom the case. It does not account for change. [Ref. 26:p. 37]

The purpose and objective of RAD is to provide system development which is identified with high speed, high quality and lower cost. [Ref. 36:p. 11] Some organizations using CASE tools may not realize their full productivity because initial user specifications may be held static as the technical design, coding and testing is completed. This static time may equate to several months, or even years before the system becomes operational. During this time the needs of the system may change.

RAD ensures not only that the time between design and implementation is greatly reduced, but that the user is actively involved in the analysis and design phases of the system. [Ref. 36:p. 11] The factors that comprise the RAD methodology are as follows:

a. Thorough involvement of the end user in the design of the system. [Ref. 36:p. 12]

b. Prototyping, which helps the users visualize and make adjustments to the system. [Ref. 36:p. 12]

c. Use of an integrated CASE toolset, which enforces technical integrity in modeling and designing the system. [Ref. 36:p. 12]

d. A CASE repository that facilitates the re-use of well proven templates, components or systems. [Ref. 36:p. 12]

e. An I-CASE tool set that generates bug free code from a fully validated design. [Ref. 36:p. 12]

f. User involvement in the Construction Stage. This stage is where the design of a system is finalized and built by both the users and developers. This allows for details to be adjusted if necessary. [Ref. 36:pp. 12, 14]

RAD is made up of five distinct phases: modelling, prototyping, optimization, integration and deployment. [Ref. 25:p. 29] The modelling phase includes the creation of enterprise models, entity-relationship diagrams, and functional decomposition models. Prototyping consists of small development teams, usually four to seven highly skilled programmers who interface with the users of a system, and build prototypes that are continually refined until a system is ready for implementation and deployment. [Ref. 24:p. 10] Optimization is when the system has been configured for a specific environment, e.g., a payroll system, taking into account the requisite database configuration, network protocols, and hardware. [Ref. 25:p. 29] Integration is when the system is ready to operate in conjunction with other systems. Deployment is when the system is complete and ready to be used by the end user.

The benefits of using RAD are straight forward. Time is money. End users of a system can start experimenting with application prototypes from the onset of development. [Ref. 25:p. 29] This allows any errors or misconceptions about the development of a system to be corrected early. According to William Baker of James Martin & Company, RAD allows a program to be broken down to small segments, each segment is limited

to around 1000 function points. [Ref. 24:p. 10] This allows the development team more flexibility and control.

The premise of I-CASE is that it can combine the functionality of both upper and lower CASE tools, under one framework. I-CASE tools enforce a development methodology. Concerning methodologies, it should be noted that:

> . . . if you use dataflow diagrams for analysis and object-orientation for design, you change the development paradigm, requiring new information structures and formats. Some CASE systems deal with such incompatibilities by using bridges to automate the exchange of information among tools. However, these bridges obscure the basic problem--the need for a rigorous, integrated development methodology to ensure the success of integrated CASE. [Ref. 32:p. 69]

F. INTEGRATED ARCHITECTURE

For a CASE product to be considered fully integrated, it must consist of **horizontal**, **vertical**, and **cross-enterprise** integration. [Ref. 18:pp. 7-9] Horizontal integration maintains integrity within each life cycle stage. It is based upon data, activities associated to data, and interaction of data. [Ref. 18:p. 8] A key aspect of horizontal integration is that each instance of an entity has only one unique definition that is shared by all the tools that comprise an I-CASE tool set. [Ref. 18:p. 8]

Vertical integration maintains consistency and integrity to data from one stage of the development cycle to the next. This is achieved by a tight coupling between separate tools within an I-CASE suite. [Ref. 18:p. 8] For example, the data represented in an entity-relationship diagram created in the

design phase will not loose its meaning when it is moved into the production/building phase of an application development.

Cross-enterprise integration ensures that the data definitions are consistent, and that data is shared throughout an organization. This is achieved through the use of the I-CASE repository. [Ref. 18:p. 9]

G. I-CASE BENEFITS

Increased developer productivity and higher quality structured systems are the two major benefits of employing an I-CASE tool. I-CASE tools force users to adhere to methodology standards. This gives the development process the discipline required when addressing the myriad complexities of both software development and re-engineering. The ability of I-CASE tools to provide rapid prototyping enables system developers and users the advantage of uncovering early flaws in a system. This leads to better and earlier requirement definitions.

I-CASE will not necessarily produce systems overwhelmingly faster. Organizations typically save only 20% in overall development time. [Ref. 39:p. 9] And for the first few developments there may be no savings in time. However, the long-run savings is expected in the maintenance life cycle phase. Organizations have experienced as much as a 69% reduction in maintenance expenses for successful I-CASE projects. [Ref. 39:p. 9] Well structured systems require less maintenance and enable future modifications with minimal

complications. Table 3-1 displays the cost of adding enhancements.[19] It clearly shows that a well structured system can undergo enhancement modification in less time and with less cost. The important thing to consider from this table is that both CASE and I-CASE tools can achieve well structured programs. In the long run, well structured systems will translate into less maintenance requirements for a system.

H. I-CASE LIMITATIONS

While there are many benefits, I-CASE should not be viewed as a panacea. The goal of I-CASE is to eventually bring different vendor tools together within a single integrated environment. The four leading I-CASE vendors, Texas Instruments, CGI Systems, Arthur Andersen and Knowledgeware, all have I-CASE products that can work with other third party CASE and I-CASE tools to a degree. But here is where the difficulty lies. For example, Knowledgeware's I-CASE tools Information Engineering Workbench and Application Development Workbench (IEW/ADW) may be able to take information from a different vendor such as Texas Instruments IEF. However, in so doing, you will usually loose functionality of the information. This is because each tool models data differently. [Ref. 49] Both I-CASE tools are similar in that

---

[19] Table 3-1 reflects data collected by Software Productivity Research, Inc., and presented at the National Re-engineering and Maintenance conference on August 10, 1992 in San Jose, California.

they support the information engineering methodology. [Ref. 50] However, IEW and ADW can be combined with other methodologies and work better with different vendor tools. [Ref. 50] A major difference between these two I-CASE tools is that Texas Instruments IEF provides a rigorous, enforced methodology (Information Engineering) within a tightly controlled environment, Knowledgeware's I-CASE tool does not. [Ref. 50] "Knowledgeware's products lack the high level of integration intrinsic to IEF." [Ref. 50]

No vendor has yet produced a framework that can fully integrate different third party CASE or I-CASE tools. There are two limitations that hamper inter-vendor integration. First, single vendor I-CASE tools, such as Texas Instruments IEF, are proprietary and lock a user into a single architecture. [Ref. 16:p. 31] Second, there are no current standards available for industry to follow. However, there are current tool integration standards efforts underway.[20] Other standards initiatives:

> . . . include government-backed, industry, and ad hoc standards efforts aimed at data management, tool portability, tool integration, and tool architecture. Among these efforts, no single standard is likely to supersede all other standards and independently guarantee future environment integration. [Ref. 37:p. 27]

---

[20] Some of these CASE specific standards include: CASE Data Interchange Format (CDIF), Portable Common Tools Environment (PCTE), CASE Integration Standard (CIS), and A Common Tool Integration Standard (ATIS).

In addition to the lack of integration frameworks and standards, a major limitation of I-CASE is how organizations employ it. I-CASE tools constitute a major financial investment. It takes time, commitment to organizational change, and qualified personnel to effectively use I-CASE.

I. THE DOD I-CASE PROCUREMENT

The DoD has a large inventory of software applications. Many of these applications are old and unresponsive to changing requirements placed on them. They also require high maintenance costs. DoD MIS software applications run on approximately 160 large mainframe computers, 400 mini-computers and over 250,000 IBM MS-DOS compatible personal computers. [Ref. 31] The main programming languages used in these applications are COBOL 74, C, FORTRAN, Pascal and 4th generation languages. [Ref. 31] Few of these applications are portable across different hardware platforms. The main problem facing the DoD is that many of its systems are developed over multiple hardware platforms, operating systems, and programming languages. This has created redundant applications that cannot be shared among government agencies. [Ref. 31] Since DoD's MIS applications are not portable to open systems hardware environments, they can only run on the proprietary hardware platforms in which they were created. This has complicated DoD's training requirements and hampered efforts to reduce costs and increase productivity. [Ref. 31]

To overcome the complications above and improve productivity in developing, maintaining and re-engineering its MIS systems, the DoD is pursuing an I-CASE procurement. The I-CASE procurement seeks to maximize the use of commercial off the shelf (COTS) components and to enable the rapid production of portable Ada software applications that comply with National Institute for Standards and Technology (NIST) Application Portability Profile standards. [Ref. 31] The objectives of the I-CASE procurement are:

a. Improve software quality and productivity while reducing the cost and risk associated with the development of MIS software systems by establishing a standard software engineering environment that supports a formal, repeatable software development process throughout the software development life cycle. [Ref. 31]

b. Reduce software development and maintenance costs as well as reducing the time required to respond to changing user requirements. [Ref. 31]

c. Establish and provide a standardized, software engineering environment that provides a fully integrated set of Commercial-Off-The-Shelf (COTS) components supporting the entire life cycle. [Ref. 31]

d. Establish a software development environment that supports the development of portable applications that execute on open systems platforms, and reduces the amount of source code that must be manually generated to develop a CIM software application. [Ref. 31]

e. Provide an environment that will incorporate the reuse of domain knowledge and source code to eliminate manual source code development and improve software productivity. [Ref. 31]

f. Provide an environment that supports a re-engineering process for converting large MIS applications to an Ada/Relational Database Management System (RDBMS) implementation. [Ref. 31]

g. Provide for the training and education of software development personnel in the use of the environment and the software development process supported by the I-CASE environment. [Ref. 31]

The I-CASE procurement is a seven year contract that has an additional three years for maintenance requirements. The procurement requirements are divided into three tiers. The first tier requirements are those that can be met with existing technology. These can be considered the minimal mandatory requirements. The second tier are requirements that are not wide spread within the industry, but can be demonstrated. These are more specific requirements that potential vendors must be able to demonstrate. The third tier is a migration plan for new COTS tools to be migrated into the DoD I-CASE tool. The migration plan is unique in that it takes into account that as technology evolves with I-CASE, tier one and tier two will not satisfy all the I-CASE requirements. The COTS requirement prevents vendors from providing I-CASE tools exclusively for DoD. Therefore, if a product is upgraded, DoD will not be isolated with tailored I-CASE tools. [Ref. 51] This contract's Request for Proposal was released to industry in August, 1992. Contract award is projected for May, 1993.

J. SUMMARY

This chapter has examined a framework that identifies and explains the elements for software tool integration. This chapter also covered the distinction between CASE and I-CASE

tools, the importance of a repository, and a review of two methodologies used with I-CASE. A comparison between two I-CASE tools, IEF and IEW/ADW, was discussed to highlight the benefits and limitations of I-CASE. The DoD I-CASE procurement was briefly discussed and the objectives that DoD anticipates with I-CASE covered. Re-engineering is one of the principle objectives.

One of the key items stressed in this chapter was the need not only for a methodology to enforce procedures, but that using I-CASE requires highly skilled and trained people. Otherwise, the potential exists for development teams to only create bad systems faster. Speaking at the CASE World Conference & Exposition on February 18, 1992, Ed Yourdon commented that "I-CASE tools do not make people smart--smart people use I-CASE." I-CASE tools cannot address every kind of application, build systems for every type of hardware environment or use every type of database. [Ref. 29:p. 10]

Having looked at the benefits and limitations of I-CASE, the following chapter will summarize data collected from an organization using an I-CASE tool in a re-engineering capacity.

IV. RE-ENGINEERING WITH I-CASE IN DoD: DATA COLLECTION

During the course of this investigation, civilian and military organizations that were using I-CASE tools in a re-engineering capacity were contacted for data. Additionally, CASE/I-CASE vendors, research consultants, DoD research facilities, and academic personnel were contacted to provide background information on re-engineering and I-CASE theory. The author also attended one CASE conference and one re-engineering conference that provided information on the latest trends in re-engineering and CASE technology.

Having completed an extensive review of the literature available on re-engineering, CASE and I-CASE, the next phase of the research was to see how an organization was using an I-CASE tool. Three DoD activities were identified and sent a questionnaire. However, only one of these activities responded. Even though the information obtained through the questionnaire and phone conversations with the facility did not reveal statistically relevant data, the responses nevertheless contained valuable information that helped in understanding how an organization adapted, learned and used an I-CASE tool for re-engineering.

A. DATA SEARCH

Upon the onset of this research, collecting data appeared to be simple and benign. This initial view soon faded. With

the exception of Texas Instruments, most I-CASE vendors were reluctant to reveal clients that were using their products. This was because many of these vendors had non-disclosure agreements with their clients to ensure confidentiality. Military installations were somewhat more receptive. However, with what the researcher perceived to be sensitivity surrounding the current DoD I-CASE procurement, some military organizations were hesitant to disclose too much information regarding their own re-engineering efforts. The remainder of this chapter is devoted to the discussion of the results obtained from a questionnaire and numerous phone conversations with the one DoD activity that had been using an I-CASE product for re-engineering a 250,000 (LOC) COBOL program.

B. INQUIRY BACKGROUND

The military organization that agreed to share information on their re-engineering efforts is located on the east coast. A questionnaire was developed and mailed to the facility. Friendly and cooperative bilateral communication was established through phone conversations with this facility. This was useful in clearing up any misconceptions and problems with the questionnaire, and provided an avenue to collect additional data. Appendix B is a copy of the questionnaire.

C. INFORMATION SOLICITED

The objective of the data collection effort was to obtain information that would provide an indication as to how an

organization is adapting and using an I-CASE tool for a software re-engineering project. The questionnaire, plus data collected via phone conversations, specifically focused on the following attributes:

1. Re-engineering: what was the initial state of the original system's source code and documentation? Did the organization have data administration policies in place that set standards?

2. What type of training was provided for the users and what were the lessons learned from the training?

3. Learning curve: how long did it take the users to become acclimated and proficient in using an I-CASE tool and its associated methodology? Was there any resistance to using I-CASE?

4. Performance: did the I-CASE tool meet the expectations of the users? If no, what areas were deficient? If yes, in what areas was the tool superior?

D. DATA RESULTS

The results from the questionnaire provided a useful example for information as to how a development team adapted and employed an I-CASE tool. The answers associated with the four attributes mentioned in the previous section are discussed in the following sections.

1. Re-engineering

The organization is using an I-CASE tool for re-engineering one system that had poor and out of date documentation. The original system consisted of 48 entities and approximately 250,000 lines of source code. The source code was characterized as being unstructured "spaghetti" code that had been modified numerous times over its 15 year life.

Function point analysis was used only to a small degree to determine the complexity of the source code. Surprisingly, this facility does not have a set data administration policy. They indicated that this is an area that needed to be addressed.

2. Training

Twelve people were initially chosen for training. The organization began training with third party vendors and consultants to learn methodology, business area analysis, database training, strategic planning, code construction and other areas relevant to an I-CASE environment. Even though this training was less expensive, the organization felt that the training was not adequate. It was determined that training should be sought from the vendor of the I-CASE tool being used. This action was taken. The 12 members that attended the initial training from third party vendors and consultants also went through training provided from the I-CASE vendor. Even though the I-CASE vendor training was more expensive, it was determined that the level of training was superior than what third party vendors or consultants could provide. The training consisted of five classes. It was estimated that each of the five classes cost $1,200 per person. The lesson learned was that the I-CASE vendor should Have been sought from the start to provide training.

3. Learning Curve

With the benefit of the on-site schools plus hands on experience with the I-CASE tool, it was estimated that it took six months for a person to become fully comfortable and proficient with the I-CASE tool. Initially some of the "older" personnel on the development team were resistant to using the I-CASE tool, especially with the concept of code generation. As the team progressed and became more proficient, the initial resistance soon faded. In fact, as the project nears its completion date of October 1992, the entire development team has become "sold on" I-CASE. There were no concepts or areas identified as being difficult to become proficient in. The 12 people selected for training were hand picked and considered the most qualified for the training. As the initial 12 members completed their training and began working with the I-CASE tools they taught other members within the organization.

4. Performance

There were no comments indicating that the I-CASE tool was deficient in its performance. However, it was noted that it took a considerable amount of time to accomplish tasks with the I-CASE tool. This was because the I-CASE tool being used (Texas Instruments IEF) requires strict adherence to procedures in its methodology. IEF uses the information engineering methodology. The DoD activity had not previously used this methodology. One of the team leaders commented to

the author that this was not necessarily a limitation. It took time because there are many procedures to learn and become proficient in. Coupled with the time needed to learn the methodology and the procedures for using the I-CASE tool, other factors to consider are the size and complexity of the application being re-engineered. Besides the comment on the time expended on a task, the I-CASE tool received high marks on performance.

E. SUMMARY

When asked about meeting deadlines for the project, the questionnaire revealed that 95% of the schedule deadlines were met. The inability of not meeting the other five percent was attributed to procurement problems of not being able to obtain the needed tools on time. One of the interesting things observed from the phone conversations with this facility was the utmost confidence in the I-CASE product being used. But this facility is an isolated case. The results of the questionnaire should not be viewed as a barometer for all re-engineering efforts with I-CASE. However, the questionnaire and the data obtained through phone conversations was able to provide some insight into an organization's experience using an I-CASE tool.

V. CONCLUSIONS AND RECOMMENDATIONS

This research has investigated the theory and use of I-CASE tools in a re-engineering environment. While results from the questionnaire represent a limited view of I-CASE utility for re-engineering, they have common traits with data collected from phone conversations, seminars, and electronic mail correspondence. The remainder of this chapter will focus on the conclusions, answers to research questions, lessons learned, and recommendations derived from this research.

A. CONCLUSIONS

Some of the literature and discussion surrounding I-CASE can be misleading. Theory and practicality have a way of being blurred if one does not pay careful attention to the actual capabilities of I-CASE tools. For instance, the theory behind I-CASE is that it can cover the entire software life cycle. This may be true. But it does not automate the entire life cycle process for existing systems not developed with the tool! An old COBOL program cannot simply be loaded into an I-CASE tool, and with a few key strokes, a new and improved COBOL program is reborn.

I-CASE tools are only as good as the information that people put into them. [Ref. 52] One may expect when re-engineering into an I-CASE environment that significant manual intervention will be required. Skilled people will initially

have to go over the old program line by line in order to obtain preliminary information of the structure of the program. It should be remembered that the "A" in CASE and I-CASE means "aided."

There was a common characteristic between the data collected with questionnaires and the contacts with I-CASE vendors, consultants, academicians, and users. Software re-engineering using I-CASE requires skilled and motivated people. It will take several trained individuals using an I-CASE tool to develop or re-engineer a system. Individuals must be motivated to overcome initial failures and setbacks. One should not perceive re-engineering a software program as a quick or inexpensive process. It is neither. But if properly done it offers sizeable cost savings over a system's life.

If inadequately trained and unmotivated people are using an I-CASE tool, the chances of success are slim. The organization will eventually realize that they are only producing lousy systems faster. This will eventually result in increased maintenance costs. In such cases, the utility of the I-CASE tool will have been of little value.

In the re-engineering case study conducted by the NIST and the IRS, cited in Chapter III, it was concluded that:

> Performing re-engineering requires a highly trained staff with experience in the current and target system, the automated tools, and the specific programming languages involved. Application system experts must be involved throughout the re-engineering process; they are essential for design recovery. Software engineering is a complex and

difficult process. The success of an organization's application of this technology will be determined by the level of commitment made by the organization. [Ref. 38]

The corollary to inadequate use of I-CASE is that it can produce better and enhanced systems and achieve savings both in development and maintenance. But two paramount items must be in place to assist an organization that chooses to use an I-CASE tool in a re-engineering capacity:

1. Senior management must endorse the establishment of goals. Support of a project means more than being vaguely aware of what re-engineering and I-CASE technology can accomplish. To set priorities, establish goals and make sound decisions, senior management must be educated in the software development process and the capabilities and limitations of I-CASE; and

2. An established, functioning data administration policy is required. An organization must have a means to standardize its data administration, i.e., policies that set requirements for creating, controlling and maintaining data. This prevents the duplication of data.

Miracles should not be expected overnight with software re-engineering. But with thorough planning and pro-active support from both management and technical personnel, software re-engineering can provide beneficial systems.

B. ANSWERS TO RESEARCH QUESTIONS

In this section, answers to the research questions stated in Chapter I are presented.

1. From DoD's standpoint, what needs to be considered, as well as avoided, in re-engineering its inventory of

systems within an integrated computer aided software engineering (I-CASE) environment?

According to Dr. Bill Curtis of the Software Engineering Institute at Carnegie-Mellon University, the first thing to look at is what types of systems are candidates for re-engineering. CASE and I-CASE are best suited for management information systems (MIS), not embedded real-time weapon systems. [Ref. 53] Embedded weapon systems are primarily written in assembly language and require extremely fast processing times. MIS applications' are primarily transaction processing systems and do not have as near the time critical operational requirements as embedded weapon systems.

Dr. Curtis stated there are two important issues that DoD should consider for moving into a re-engineering environment using I-CASE. First, management must plan, track, and control the re-engineering process. [Ref. 53] The infrastructure must be in place that integrates the actions and talents of both technical and managerial personnel. This will set the stage for moving to an automated environment. Second, for what ever system is under consideration, it is important to determine what data is to be captured, and how well can that data be structured in order to build the data model. [Ref. 53]

What should be avoided? Quite simply, attitudes. DoD should not build too many expectations that I-CASE is here to

answer its problems with software. As mentioned throughout this thesis, I-CASE tools are aids; people are the critical element in the re-engineering process.

2. What are the current problems facing the CASE and I-CASE industry?

The major problem facing the CASE and I-CASE industry is that there is no current integration standard to completely integrate the various CASE and I-CASE tools together under one framework. While IBM's AD/Cycle and Digital's Cohesion are attempts to offer an integration framework, there are still problems. For instance, a developer mixes CASE tools, and then changes a design element in one tool, the change may not cross over and be updated by a different tool. Another problem in this area can arise when one vendor upgrades its product; other vendors' tools may not be fully compatible with the new upgrade. While vendors strive for compatibility with their products, the wide variety of other vendor products, which are also undergoing continuous change, almost always mean there will not be full compatibility with these other vendor products. I-CASE tools, produced by a single vendor, do not have this problem. I-CASE tools and data repositories are still proprietary products and lock a user into a specific tool. A second problem facing the CASE industry is that as CASE and I-CASE technology evolves, smaller, and less influential companies will either fold or be bought out by larger and more powerful

companies. This has already happened. DoD should take into consideration a vendor's business future when assessing contractual commitments.

3. Can re-engineering using I-CASE tools produce viable systems for DoD?

Since empirical evidence on this issue for the commercial market has yet to be gathered, it was not unusual to not find data that could conclusively answer this question for DoD. However, there are examples of successful re-engineering projects that have migrated to an I-CASE environment in the civilian market. This should encourage the DoD to pursue such projects. There are certainly many old, maintenance intensive systems in the DoD inventory that are still of critical importance. The planning should begin today to identify likely candidates and initiate the management and organizational support such projects will require for success. Thus, when the DoD I-CASE tools are delivered, re-engineering of selected projects could commence. The major factors that can increase the likelihood of success, as discussed in this paper, are that of pro-active management, using the most skilled people, and a structured, disciplined methodology with I-CASE.

4. How many systems within DoD warrant re-engineering?

This question was asked of the Standard Systems Center at Gunter Air Force Base, the Software Technology Support Center at Hill Air Force Base, and the Software Engineering Institute at the Carnegie-Mellon University. The answer from all three locations was the same: no idea. In a phone conversation with a source at the Software Engineering Institute, it was disclosed that to their knowledge, no data has been kept on this issue, and therefore no accurate number could be ascertained. The fact is that today there are processes that can be used to determine which software systems should be candidates for re-engineering. DoD should begin this process now.

5. What are the estimated cost savings DoD can anticipate by re-engineering some of its applications?

This question was addressed to the same locations mentioned above, as well as individuals in the military and industry software arena. Again, since there is scant or no data kept on applications that warrant re-engineering, no approximate figure could be reached. The two best answers received from this question ranged from "use chicken bones or goat entrails to provide a figure," to a more refined, but nevertheless respected reply from the Software Engineering Institute, that it would take several man months of effort to arrive at an answer. According to a source at the Software Engineering Institute, no work has been conducted in this

area. There may be efforts currently seeking an answer to this question. However, no source could be found during this research effort.

C. LESSONS LEARNED

The most difficult aspect of conducting this research was never having been part of a development team that used a CASE or I-CASE tool. However, the author was able to attend two seminars, visit some I-CASE vendors, and work with tutorial versions of one I-CASE product. The lack of any hands on experience did not hinder the quest for information, though at times some naive and novice questions were asked. Most sources contacted were extremely cooperative and understanding. The questionnaires provided useful information, but on site and face-to-face interviews would have helped gain more insight and understanding.

The I-CASE vendors were helpful in explaining and demonstrating their products, but were not willing, with the exception of Texas Instruments, to provide clients who had used or were in the process of using an I-CASE tool. This limited the chance to obtain on site face-to-face exposure of I-CASE usage. Several CASE consultants indicated that they did not know of any published re-engineering case studies/lessons learned. This could be attributed to the fact that re-engineering and I-CASE are relatively new technologies and little information is available for dissemination. On

the other hand, some organizations may not wish to publish their shortcomings, failures, or successes.

One of the interesting aspects of the research was that re-engineering had a different meaning depending on who was asked. Chapter II of this paper discussed the most widely accepted view and definitions of re-engineering. However, one individual interviewed that was using an I-CASE tool, considered migrating components of a program into a corporate database as re-engineering. In this case, the program itself was not being re-engineered, but since minor modifications were made to the program before it was placed into a corporate database, it constituted re-engineering. The bottom line is that, for some, any change or modification, however slight, can constitute re-engineering.

The use of e-mail (electronic mail) was invaluable in the research. From the onset, e-mail was used to contact sources to help clarify issues, ask for additional sources and more important: to ask questions. Phone conversations help, but people are not always available. E-mail provided great flexibility in collecting data. It is highly recommended for anyone wishing to conduct research to use e-mail.

D. FINAL THOUGHTS AND RECOMMENDATIONS

One of the items stressed in this thesis was the need for an organization to conduct a self assessment of its software inventory to evaluate the need to re-engineer. DoD should develop guidelines in this area. The Software Technology

Support Center at Hill Air Force Base has made strides in this area. The following steps are recommended for the DoD to consider with respect to re-engineering with I-CASE tools. Some are unique to re-engineering. Others are applicable to any software project.

1. Use metric analysis tools to assess applications for re-engineering. Metric tools will allow for the identification of trouble spots within a program. This will help in determining whether a program is structured or not.

2. Start with selected pilot projects of applications identified in Step one as good re-engineering candidates. One alternative to expedite this procedure is to hire experienced consultants that can provide guidance in employing re-engineering methodologies that have been successfully used in the civilian arena.

3. I-CASE tools have been designed and used primarily for systems development. Re-engineering can be done with I-CASE tools, but it requires that people rigorously identify what data is to be captured and structured into a data repository. Much of this process is manual and requires motivated and skilled personnel to complete. Non-integrated CASE tools may also be required.

4. It is essential that managers with both technical and interpersonal skills be placed in re-engineering projects using I-CASE tools.

5. Select the most motivated and technically proficient personnel for initial training. If possible, all personnel assigned to a re-engineering project should receive training. If it is impossible to train everyone, the personnel that are trained and proficient with I-CASE tools can teach others within the organization.

6. Reward and recognize personnel for their work.

It is the responsibility of management to evaluate the goals that an organization should strive to meet. Software

re-engineering should not be attempted for its own sake, but rather in terms of the organization's goals. If a system is still of value to an organization, then re-engineering may be an option. However, some systems are beyond help because of their complexity, poor documentation, and unstructured design. If this is the case, it is better to develop a new system.

APPENDIX A

THE RE-ENGINEERING CANDIDATE SELECTION PROCESS

The candidate selection process consists of determining the software system's complexity, importance, and longevity, and then choosing the appropriate strategy calculated to be the most cost effective.

The information gathering process consists of answering a series of questions, supplying the requested metrics, and deciding whether the answer corresponds to one of three values:

- Low, medium, or high (for complexity and importance)

- Short, medium, or long (for remaining system life).

Instructions are provided in each section on determining a consensus value. More precise definitions for the terms are given later within this methodology.

Complexity Analysis of the Candidate Software

Answer all the questions in this section that you can. If you do not know the answer, attempt a consensual answer. It is strongly recommended you consult with several people when answering these questions to help minimize potential error or bias. Since each question varies in importance, each question is weighed. Multiply each answer by the weighing factor.

1. How many executable lines of code exist? (wt. = 2)
   1 - Less than 15K
   2 - Between 15K and 100K
   3 - More than 100K

2. What is the statistical mode (see the glossary) of executable lines per module? (wt. = 2)

   1 - Less than 50
   2 - Between 50 and 200
   3 - More than 200

3. What is the statistical mode of the Cyclomatic complexity per module? (wt. = 3)

   1 - Ten or less

2 - Between 10 and 20
3 - More than 20

4. What is the statistical mode of the Essential complexity per module? (wt. = 3)

   1 - Five or less
   2 - Between 5 and 10
   3 - More than 10

5. What is the system's language level? (wt. = 1)

   1 - "4GL" (advanced, user-friendly languages)
   2 - "3GL" (higher order languages)
   3 - "2GL" (assembly language)

6. The system was created using a development strategy that was: (wt. = 3)

   1 - Clear, concise, and complete
       (such as DoD-STD-2167A)
   2 - Vaguely understood
   3 - Non-existent

7. How many programming languages does the system use? (wt. = 2)

   1 - One
   2 - Two
   3 - More than two

8. Over the last 6 months, has the number of errors appeared to: (wt. = 3)

   1 - Decrease
   2 - Level
   3 - Increase

9. What is the system's age as measured from the first release? (wt. = 1)

   1 - Less than 2 years
   2 - Between 2 and 5 years
   3 - More than 5 years

10. How often is the system modified (per month)? (wt. = 2)

    1 - One or fewer times
    2 - About 2 or three times
    3 - More than 3 times

11. How many versions have been released since the system was first designed or last re-engineered? (wt. = 1)

    1 - Two or less
    2 - Between 3 and 5
    3 - Six or more

12. How many people have update access to this software system? (wt. = 1)

    1 - One or two
    2 - Three or four
    3 - More than 4

13. Does the system have a maintenance backlog? (wt. = 3)

    1 - No
    2 - Yes, But steady or decreasing
    3 - Yes and increasing

14. How many maintenance programmers know the entire system very well? (wt. = 2)

    1 - Three or more
    2 - One or two
    3 - Nobody

15. What is the percentage of maintenance personnel turnover (per year)? (wt. = 2)

    1 - Less than 5%
    2 - Between 5% and 30%
    3 - More than 30%

16. How many hours are required to maintain the system per month? (wt. = 3)

    1 - Sixteen or less
    2 - Between 16 and 32
    3 - More than 32

17. Does the maintenance organization think the system's quality is: (wt. = 2)

    1 - Improving
    2 - Remaining the same
    3 - Declining

18. Do the system's users think the system's quality is: (wt. = 3)

    1 - Improving

2 - Remaining the same
3 - Declining

19. What is the average number of years experience for those maintenance programmers expected to maintain the candidate system? (wt. = 1)

    1 - Less than 3 years
    2 - Between 3 and 10 years
    3 - More than 10 years

20. Are the original developers available for consultation? (wt. = 3)

    1 - Yes
    2 - Yes, but the system is over 5 years old or the original developers are not easily accessible.
    3 - No

21. Are the programming staff members well-trained in modern software engineering techniques? (wt. = 2)

    1 - Yes, most are
    2 - Some are
    3 - None or very few are

22. What is the organization's SEI maturity level? (wt. = 2)

    1 - Three or higher
    2 - Two
    3 - One

23. Between operating systems, the candidate software system is: (wt. = 1)

    1 - Portable
    2 - Not portable
    3 - Tightly coupled (where the software system internalizes parts of the operating system--for example, embedded assembly code or system utility calls)

24. The system's documentation is best characterized as: (wt. = 3)

    1 - Complete and current
    2 - Mostly complete and current
    3 - Non-existent or untrustworthy

Compute the Average Complexity Value

Compute the system complexity value "C" by averaging the numbers associated with each answer (1, 2 or 3) as shown by the equation:

$$C = \frac{[\text{Sum of (answer * weight)}]}{(\text{Sum of weights of questions answered})}$$

If C is 1.66 or less, the overall system complexity value is low. If C is between 1.67 and 2.33, the overall system complexity value is medium. And if C is 2.34 or greater, the overall system complexity value is high.

Importance (Risk) Analysis

Answer each of the following questions unless the question does not apply. We strongly recommend you consult with several people when answering these questions to help minimize potential error or bias. Since each question varies in importance, each equation is weighted. Multiply each answer by the weighing factor.

1. If the system failed for a significant period of time, what would be the effect on the organization? (Significant is a term relative to the system being considered.) (wt. = 3)

   1 - Little or no damage
   2 - Significant damage
   3 - Permanent damage

2. How frequently does the system execute? (wt. = 1)

   1 - Quarterly or less
   2 - Weekly
   3 - On-line

3. Are there back-up systems (current, recently tested, and ready at a moment's notice) which could be used if the system fails? (wt. = 2)

   1 - Yes
   2 - Yes, but with some difficulty and a significant loss of efficiency
   3 - No

4. How much of the organization's finances does the system control or generate? (wt. = 2)

   1 - None
   2 - Some

3 - A significant percentage

5. Does the system represent a unique and important competitive advantage within the industry? (wt. = 3)

   1 - No
   2 - Somewhat
   3 - Yes

6. If the system failed, what is the potential for loss of life, lawsuits, aircraft failure, etc.? (wt. = 3)

   1 - None
   2 - Some
   3 - Significant

Compute the Average Importance Value

Compute system importance value "I" by averaging the numbers associated with each answer (1, 2 or 3) as shown by the equation:

$$I = \frac{[\text{Sum of (answer * weight)}]}{(\text{Sum of weights of questions answered})}$$

If "I" is 1.66 or less, the overall system importance value is low. If "I" is between 1.67 and 2.33, the overall system importance value is medium. And if "I" is 2.34 or greater, the overall system importance value is high.

Lifetime Analysis (Remaining System Life)

The Lifetime Analysis evaluates an existing system to determine how long a system will be maintained. The useful system lifetime is usually a management decision, but it should be based on technical aspects of the system and user expectations. Overall system health, combined with the results of the Complexity and Importance Analysis, should be used to determine this lifetime value.

To derive the lifetime value, use the above information and decide how long the system will remain active. Next, assign a value of short, medium, or long according to the following criteria:

- Short if the remaining life is 6 months or less.
- Medium if the remaining life is greater than 6 months, but less than 3 years.
- Long if the remaining life is 3 years or more.

Choose a Re-engineering Methodology

Using the three values of system complexity, system importance, and remaining system life, use the appropriate selection matrix on the following pages to determine the re-engineering methodology.

A cost analysis can be performed for all six re-engineering choices, but this re-engineering selection process provides the re-engineering choice that should be the most cost-effective for the system's overall health.

(Note: A short lifetime normally makes re-engineering impractical regardless of complexity or importance. Thus, Figure 4-1 reflects a "Leave Alone" choice.)

Candidate Cost Analysis

The purpose of the candidate cost analysis is to implement the re-engineering strategy that will best reduce monthly maintenance costs. The cost analysis is simply a comparison of the current monthly system maintenance cost against the monthly cost (pro-rated over the expected life of the system) of implementing the re-engineering strategy and the estimated maintenance thereafter.

If the re-engineering strategy is "Leave Alone" (that is, the remaining system life is short for all levels of complexity and importance) then the need for a candidate cost analysis is obviously unnecessary as this cost is the same as the current maintenance cost.

METHODOLOGY MATRIX FOR RE-ENGINEERING

Short Lifetime Remaining

Complexity	Low	Med	High
High	Leave Alone		
Med			
Low			

Importance

Figure I. Short Lifetime Remaining

Medium Lifetime Remaining

Complexity	Low	Med	High
High	Restructure	Transverse	Transverse
Med	Reformat	Restructure	Transverse
Low	Leave Alone	Reformat	Transverse (pilot project

Importance

Figure II. Medium Lifetime remaining

Long Lifetime Remaining

Complexity	Low	Med	High
High	Restructure/ Redocument	Transverse/ Redocument	Redocument/ Transverse/ Replace
Med	Restructure/ Redocument	Transverse	Transverse
Low	Reformat	Restructure	Transverse

Importance

Figure III. Long Lifetime Remaining

Maintenance Cost

The maintenance cost is the sum of projected enhancement costs, operational costs (including personnel), and failure costs. All of these figures should be readily available from previous system reports or financial statements. If not, then a close estimate must be determined.

It is important to review the maintenance cost over several years and to chart the maintenance cost (quarterly or whichever time unit best suits your organization's needs) to see if the cost is changing at a predictable rate. This is important since if the cost is rising or falling rapidly, then a charted cost will be a better predictor of future costs rather than a single figure from last month.

If the maintenance costs remain essentially constant, then the correct maintenance cost can be extrapolated from the chart. This extrapolation should be done for the entire estimated remaining system life. An average quarterly or monthly maintenance cost must be calculated to be compared with the pro-rated, average implementation cost.

Cost for Reengineering Implementation

To find the pro-rated monthly implementation cost (C_{pmi}), use the equation below:

$$C_{pmi} = \frac{\text{[Implementation cost)}}{\text{(remaining system life in months)}} + M_i$$

Here, M_i is the estimated monthly maintenance cost after re-engineering. The Implementation cost must include the costs associated with the factors below. Find the value of each (if applicable) and use the resultant sum in the equation above.

- Software too(s) expense (including maintenance contract)
- System analysis for future maintenance requests
- Implementation (system and personnel expenses)
- Any software modifications
- Additional required hardware or hardware upgrades
- Procedures modification or development
- Training
- Operating (system and personal expenses)
- Post-implementation support

M_i is calculated by taking the current monthly maintenance cost and multiplying it by one of the following estimated cost savings percentages (plus or minus 5%):

- 95% if Reformatting
- 75% if Redocumenting
- 50% if Restructuring
- 25% if Transverse engineering

Note that these maintenance costs decrease the more the system is re-engineered (if done correctly). These percentages are not necessarily the cost savings that every organization will see. But based on the experience of the authors and as reviewed by acknowledged experts, they represent reasonable values for the average re-engineering effort. Finally, the Replace cost percentage is not listed since it has no accurate value. An accurate and in-depth

analysis of system replacement is complex, and varies widely with each application. Calculating a replacement cost is beyond the scope of this report. However, this process should provide the desired results if the replacement cost is determined independently.

Analysis of Cost Results

Cost results are analyzed by comparing the current monthly maintenance cost plus the estimated impact due to system failure against the estimated pro-rated monthly maintenance cost (C_{pmi}) following re-engineering plus the costs of implementation. If either cost is significantly larger than the other, then implementation of the lower-cost option should save you money. If the costs are approximately the same, then the organization should review its priorities and objectives to determine whether re-engineering is in its best interests.

Implementation of Choice

If the re-engineering methodology has successfully passed the candidate cost analysis, then one must determine which system modules need to be re-engineered. Not every part of a candidate system need be re-engineered. Significant savings can still be accrued by re-engineering a few critical (usually labor intensive) areas of the candidate system. This strategy will concentrate re-engineering efforts and organizational resources on those problem areas.

If translation has been mandated (to Ada source code, for example), then re-engineering becomes essential. Since source code translation is not a line-for-line operation, some re-engineering will be required to accommodate the new language and its capabilities. If translation is not mandated, then considered. A modern language, applied within the context of modern software engineering techniques, can offer better tools and constructs as well as an environment conducive to greater software quality.

Next, management support must be obtained to implement the chosen re-engineering strategy. If management has been actively involved during this re-engineering economic evaluation, this step should be a mere formality. If not, management must be convinced that the re-engineering investment will be cost effective and help meet internal organizational goals. This re-engineering decision-making process (especially with the candidate cost analysis) will form the basis for a detailed study to implement a specific re-engineering plan.

Once implementation is justified and granted, the re-engineered system's maintenance costs should be periodically

compared to the estimated cost. This comparison is necessary to fine-tune subsequent analysis efforts to fit your organization's unique needs.

Glossary of Terms

Cyclomatic Complexity is a measurement of the number of paths through a program.

Essential Complexity is a measurement of the level of "structuredness" of a program.

Mode is a term from statistics denoting the most common number found in a distribution. For the complexity questions, mode refers to those modules whose size or complexity is typical.

Redocumentation is the creation or revision of a semantically equivalent representation with the same relative abstraction level.

Reformatting tools are redocumentation tools which make source code indentation, bolding, capitalization, etc., consistent.

A Restructurer is a software tool that makes source code more understandable by implementing modern programming constructs and reformatting.

Transverse engineering is the combination of reverse engineering and forward engineering, including any design changes prior to forward engineering.

APPENDIX B

RE-ENGINEERING QUESTIONNAIRE

1. How many programs/systems have you re-engineered using an I-CASE tool? What percentage of these were unstructured code?

2. How accurate was the original system's documentation?

   a. accurate and up to date

   b. moderate

   c. marginal

   d. poor - out of date

3. During your re-engineering effort, have you encountered any areas that had to be re-engineered manually?[1] If yes, which areas?

4. Provide a breakout of the original system undergoing re-engineering using the following attributes:

   a. how old is the original system_____

   b. number of batch programs_____

   c. number of interactive programs_____

   d. number of assembly programs (if any) _____

   e. total estimated lines of code_____

   f. number of computer languages used_____

5. Was metrics analysis performed before and after re-engineering, i.e., did you use a metric tool like the McCabe Cyclomatic Complexity Metric, Essential Complexity Metric,

---

[1] Manual in this sense means: was the original system so messed up, did you have to physically sit down and draw your own dataflow diagrams, entity-relationship diagrams, or write code?

Design Complexity Metric, Battle Map Analysis Tool, or any other vendor metric tool to assist you in finding areas of your program code that were revealed as error prone or hard to maintain? If yes, what type:

a. Lines of Code (LOC) count

b. Function Point Analysis

c. Cyclomatic Complexity

d. Other

6. In a previous Federal re-engineering case study conducted by the National Institute of Standards and Technology (NIST) and the Internal Revenue Service (IRS), it was determined that the complexity of the re-engineering process increased in relation to the complexity of the programs. The programs that required the most manual intervention were the programs that were also the most complex.[2]

a. Before your re-engineering process began, how was priority of programs to be re-engineered determined (i.e., metric analysis, personal experience with the system at hand etc.)?

b. Have you found a correlation between the most complex programs and manual intervention?

7. At the current point in your re-engineering effort, what percent of the re-engineering has been automated and what percent has been manual?

	Automated	Manual
a. number of batch programs		
b. number of interactive programs		

8. What type of re-engineering methodology is your organization utilizing for re-engineering, e.g., Information Engineering, Rapid Application Development, others?

---

[2] The case study cited in this question is titled "Software Reengineering: A Case Study and Lessons Learned," by Mary K. Ruhl and Mary T. Gunn. It is published by the Cutter Information Corporation, 37 Broadway, Arlington, MA 02174-5539

9. Does your organization have data administration polices set throughout, i.e., are there policies and procedures that determine how data is structured and defined?

10. How would you describe the "learning curve" of your re-engineering team in terms of time required to for the team to become acclimated and confident in the following:

a. re-engineering methodology

b. tools (I-CASE tools that incorporate dataflow diagrams, entity-relationship diagrams etc.)

c. cultural adjustment (i.e., were some personnel hesitant to learn techniques incorporated with I-CASE tools)

11. How was the time frame established for re-engineering efforts?

12. What has been the success rate in meeting the original schedule for re-engineering projects?

a. percentage that were completed before the schedule completion date_____

b. percentage that were completed on schedule_____

c. percentage that were completed after the scheduled completion date_____

13. What were the primary reasons (if applicable) re-engineering efforts have been over schedule?

14. In what areas (if applicable) has the I-CASE tool not met your expectations:

a. analysis

b. design

c. code generation

d. Implementation

e. learning curve/ease of use

f. other

15. CASE tools have been most effective in activities which actively use a structured analysis and design methodology. To what extent did your organization use such a methodology?

a. regularly (on almost all projects)_____

b. sometimes/Usually (40%-70% of the projects)_____

c. seldom/not at all_____

16. What training was provided those using I-CASE tools for re-engineering or development? (any specific schools provided by the vendor or DoD)

17. To what extent was a user involved in the re-engineering effort?

18. Do you use contractor support for the I-CASE tools and re-engineering?

APPENDIX C

LIST OF FIGURES

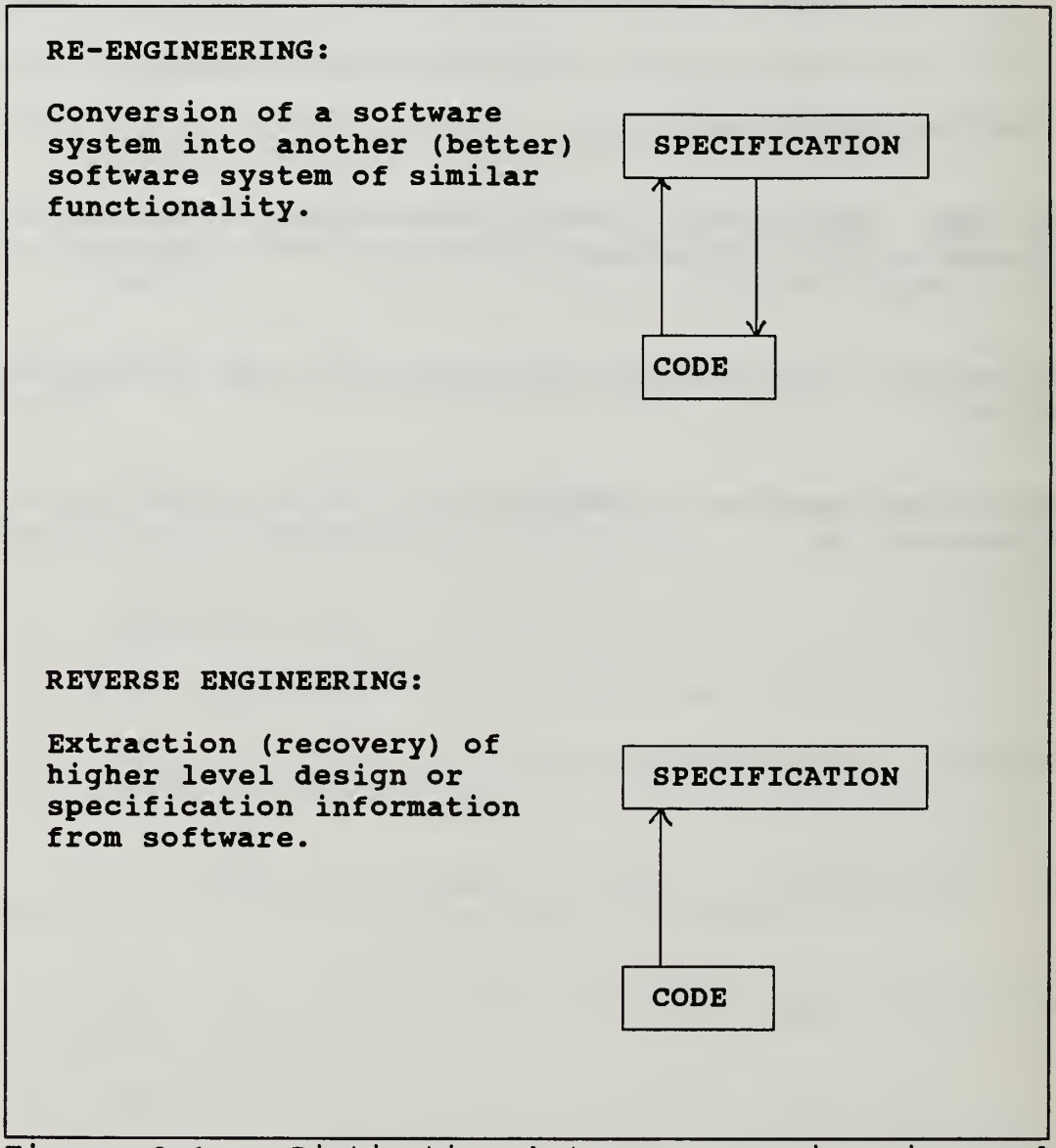

Figure 2-1. Distinction between re-engineering and reverse engineering. [Ref. 5]

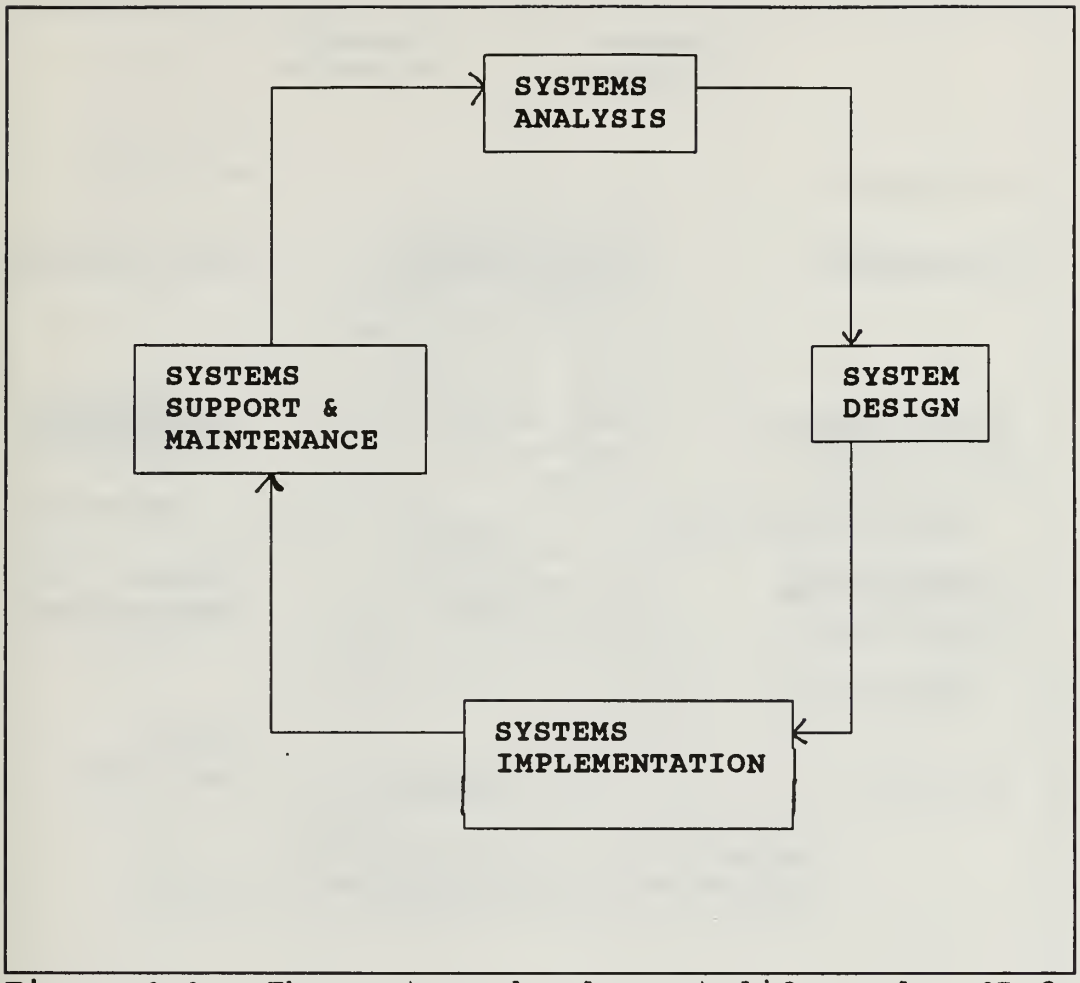

Figure 2-2. The systems development life cycle. [Ref. 21]

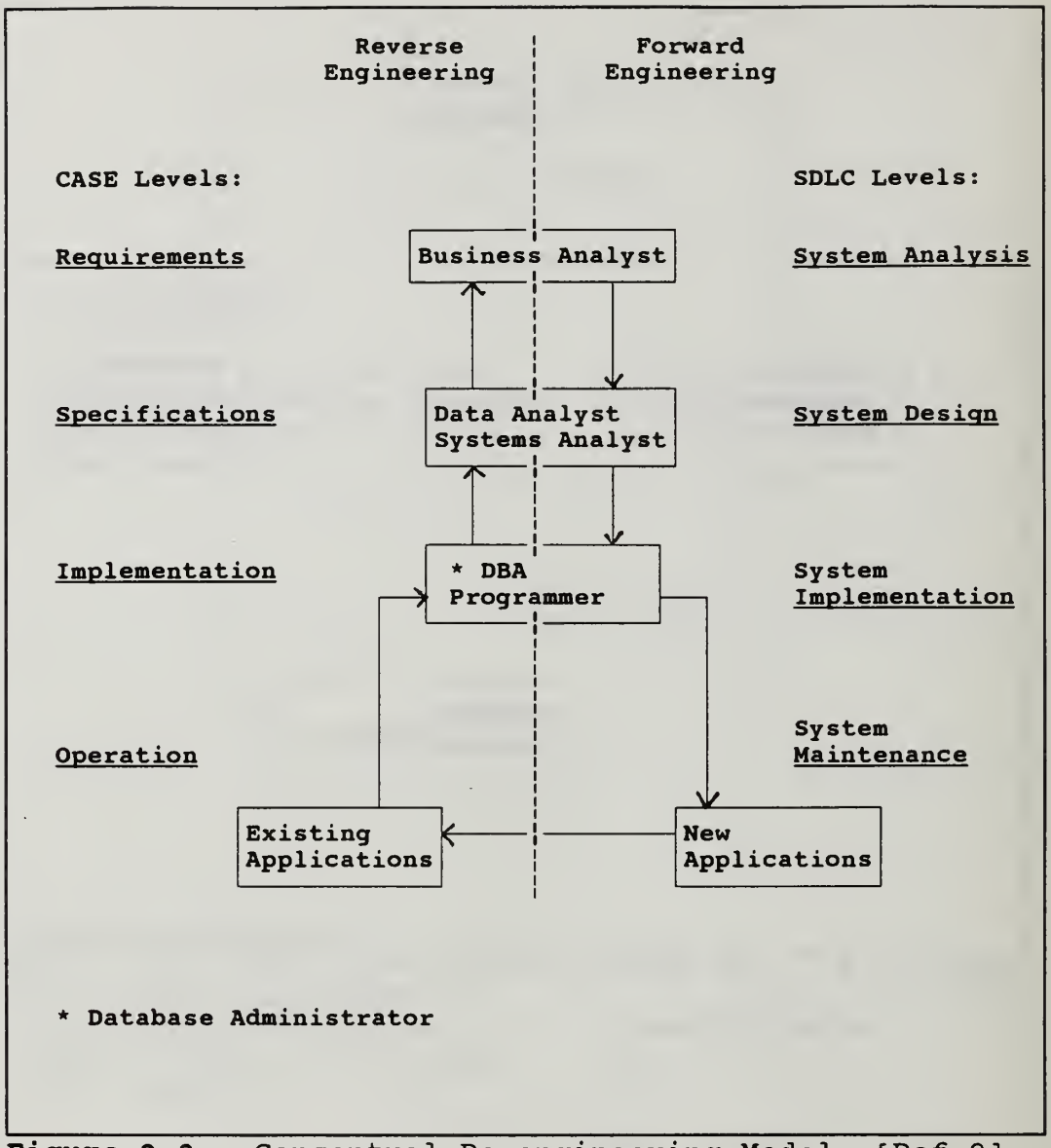

Figure 2-3. Conceptual Re-engineering Model. [Ref. 9]

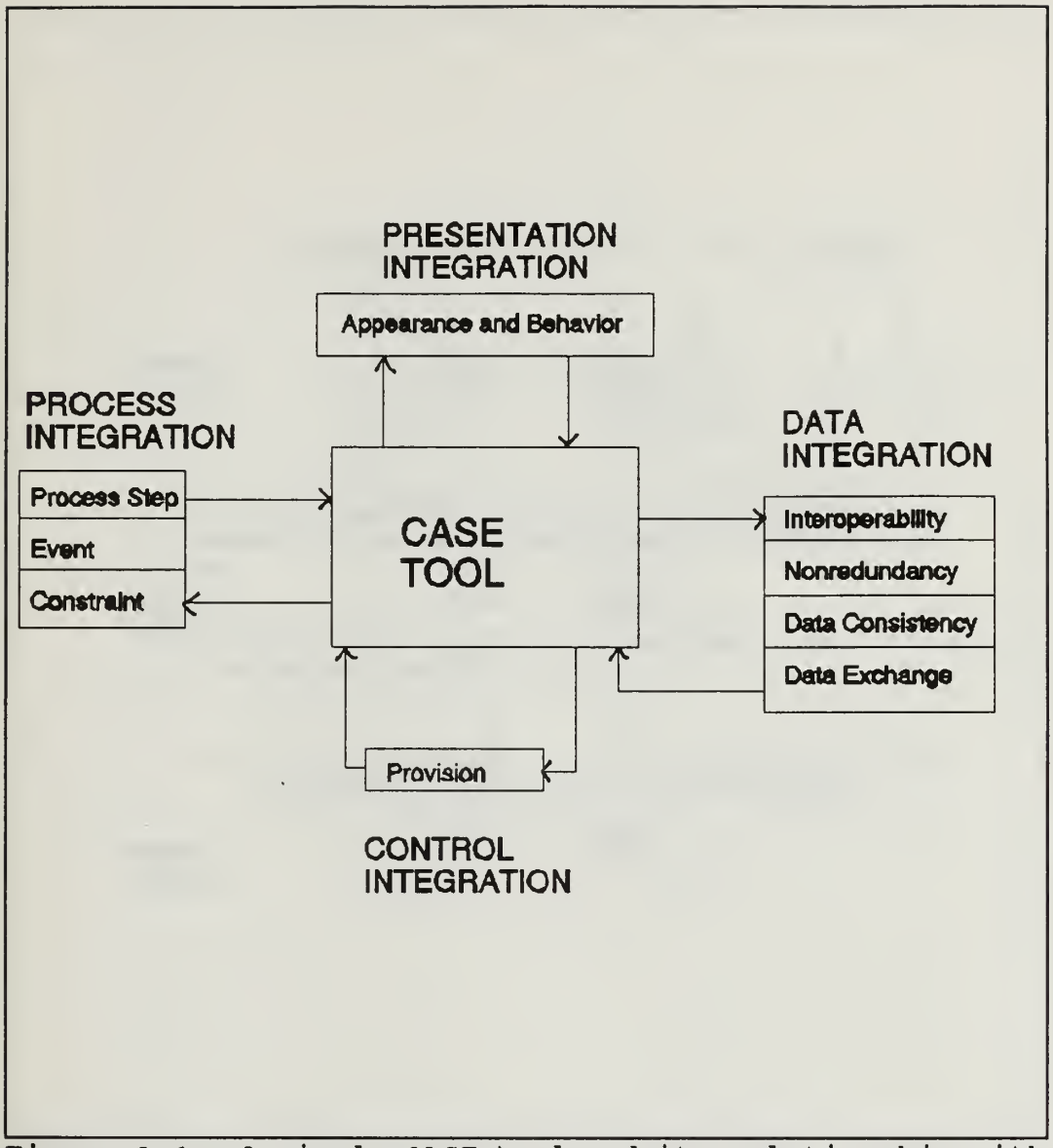

Figure 3-1. A single CASE tool and its relationship with the four types of integration. [Ref. 32]

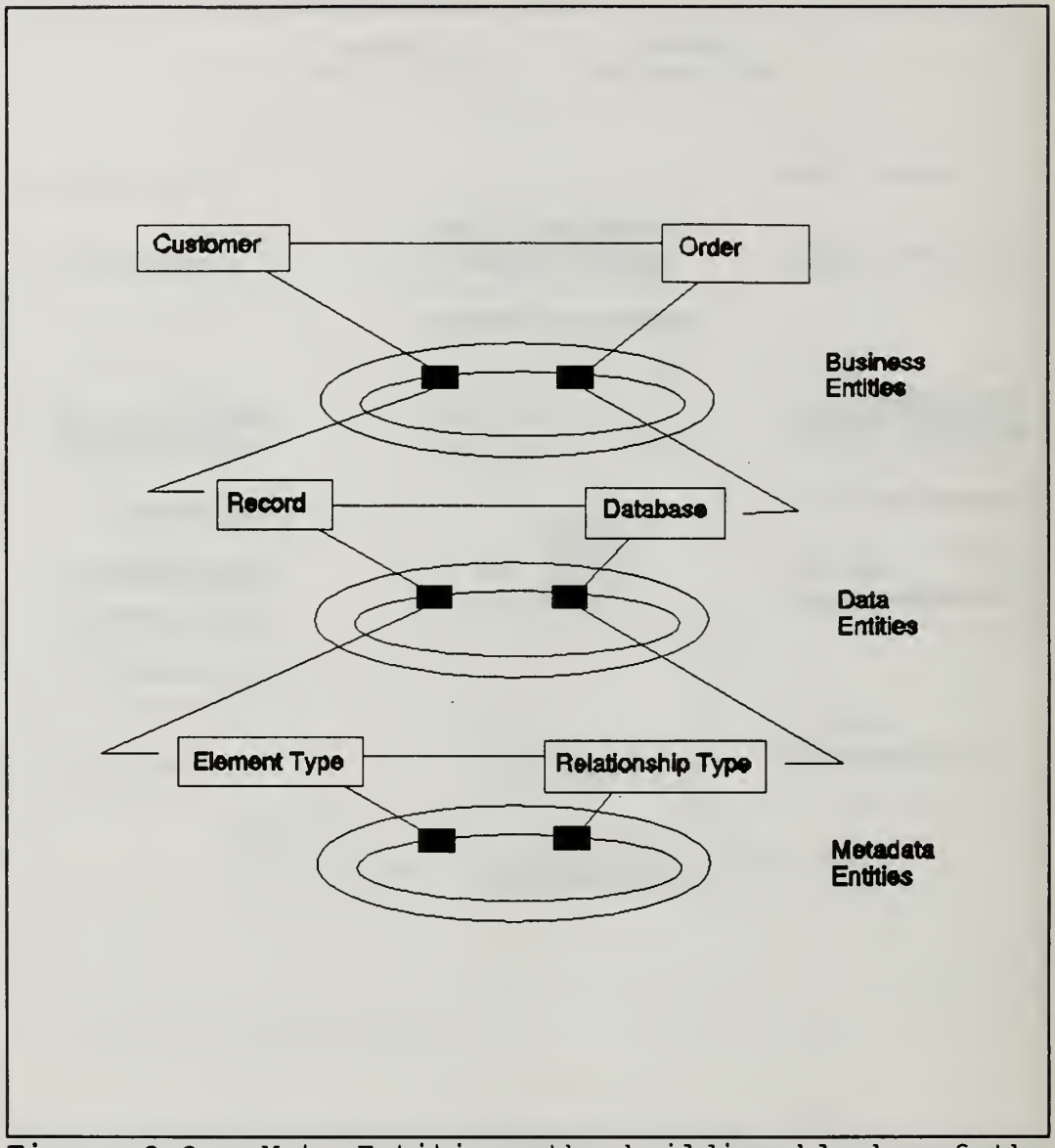

Figure 3-2. Meta-Entities, the building blocks of the repository. [Ref. 35]

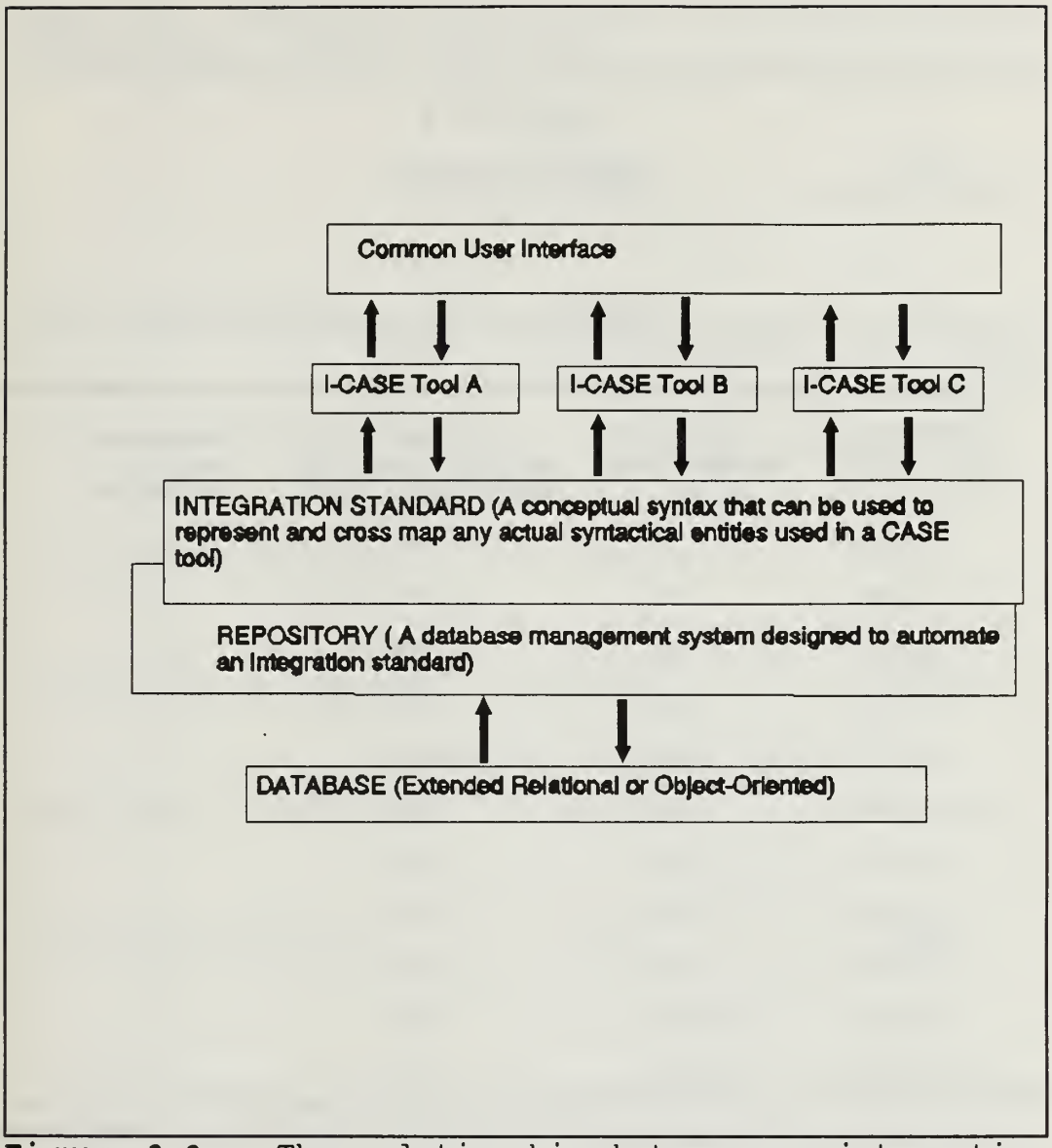
Common User Interface
I-CASE Tool A
I-CASE Tool B
I-CASE Tool C
INTEGRATION STANDARD (A conceptual syntax that can be used to represent and cross map any actual syntactical entities used in a CASE tool)
REPOSITORY (A database management system designed to automate an integration standard)
DATABASE (Extended Relational or Object-Oriented)

Figure 3-3. The relationship between an integration standard and a repository. [Ref. 17]

APPENDIX D

LIST OF TABLES

Table 2-1. COBOL ENHANCEMENT PRODUCTIVITY RATES. [Ref. 40]

BASE SYSTEM SIZE (LOC)	OPTIMAL ENHANCEMENT SIZE	AVERAGE PRODUCTIVITY RATE (LOC/PERSON YEAR)	FUNCTION POINTS/ PERSON YR.
1,000	30	16,000	160
2,000	60	12,000	120
4,000	120	10,000	100
8,000	240	8,000	80
16,000	480	6,000	60
32,000	960	5,500	55
64,000	1,920	5,000	50
128,000	3,840	5,000	30
256,000	7,680	2,000	20
512,000	15,360	1,000	10
1,024,000	30,720	500	5

Table 2-2. ENHANCEMENT CASE STUDIES: THE SIGNIFICANCE OF USING METRIC ANALYSIS. [Ref. 40]

	Poorly Structured	Well Structured
Defect Potential	250	75
Removal Efficiency	85%	95%
Defects at Delivery	38	4
Stabilization Period	5 months	2 weeks
Mean Time to Failure	1.5 hours	28 hours
User Satisfaction	low	high

Table 2-3. COMMON RE-ENGINEERING PITFALLS. [Ref. 41]

1. Resistance to change.

2. Lack of a proven methodology to guide the system re-engineering team, e.g., must be able to collect metrics and know how to interpret them. Must have a methodology from start to finish.

3. Failure to identify a target environment.

4. Failure to integrate with other system options, i.e., it may be better to redevelop than to re-engineer the system.

5. Failure to identify a business need. If a business analysis is not conducted, a reengineering tool may drive the process rather than the business needs of the organization being the driver.

6. Inadequately trained managers.

7. Lack of quality integrated tools.

8. Failure to perform an up front assessment, i.e., an organization should not jump into a re-engineering project without prior planning.

9. Inadequate education and training.

Table 3-1. POOR VERSUS WELL STRUCTURED SYSTEMS. [Ref. 40]

	Poorly Structured	Well Structured
Requirements	1 month	1 month
Design	2 months	1.5 months
Coding	4 months	2 months
Documentation	.5 months	.5 months
Integration/Test	4 months	1 month
Management	1 month	.5 month
Total Enhancement	12.5 months	6.5 months
Total Costs	$75,000	$39,000

LIST OF REFERENCES

1. Chikofsky, Elliot J. and Cross, James H, "Reverse engineering and design recovery: a taxonomy," IEEE Software, January 1990.

2. Mahon, Andrew, "Reengineering: where to begin," New Science Report on Strategic Computing, February 1991.

3. Weinman, Eliot D, "The promise of software reengineering," Informationweek, April 22, 1991.

4. MacKinnon, Peter, "CASE myths debunked," Computing Canada, Vol 17, No. 11, May 23, 1991.

5. Bush, Eric, "Re-engineering and Reality," Procedings from the Digital Consulting, Inc., CASE World Conference & Exposition, Santa Clara, California, 18-20 February 1992.

6. Ulrich, William M, "Re-engineering: Defining an Integrated Migration Framework," Case Trends, November-December 1990 to May-June 1991 issues.

7. Jones, Capers, Applied Software Measurement, McGraw-Hill, Inc., 1991.

8. Jones, Meilir Page, The Practical Guide to Structured Systems Design, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

9. Bachman, Charlie, "A CASE for Reverse Engineering," DATAMATION, Vol. 34, No. 13, July 1, 1988.

10. Ulrich, William D. and Weinman, Eliot D, "Reality of reengineering," Informationweek, December 30, 1991.

11. Snell, Ned, "Using CASE to rebuild software," Datamation, Vol 37, No. 15, August 1, 1991.

12. Koka, Ravi, "Reverse Engineering - The Missing Link," Procedings from the Digital Consulting, Inc., CASE World Conference & Exposition, Santa Clara, California, 18-20 February 1992.

13. Seymour, Patricia, "Critical Success Factors For Software Re-engineering," Procedings from the Digital Consulting, Inc., CASE World Conference & Exposition, Santa Clara, California, 18-20 February 1992.

14. Martin, James, CASE & I-CASE, High Productivity Software, Inc., Marblehead, Massachusetts, 1988.

15. Manley, Gary W., The Classification and Evaluation of Computer Aided Software Engineering Tools, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1990.

16. Burke, John P., "Tough CASE," HP Professional, Vol. 5, No. 7, July 1991.

17. Harmon, Paul, "Intelligent CASE: The Status of CASE in Early 1992," Intelligent Software Strategies, Vol. 8, No. 3, Cutter Information Corporation, March 1992.

18. "Information Engineering Facility: A Totally Integrated CASE Environment," Texas Instruments Incorporated, Plano, Texas 1991.

19. AD/Cycle Digest, International Business Machines Corporation, Second Edition, Fall 1991

20. "Management Issues: Software Engineering For Redevelopment," CASE Strategies, Vol. 3, No. 8, August 1991.

21. Whitten, Jeffrey L., Bentley, Lonnie D., and Barlow, Victor M., Systems Analysis & Design Methods Second Edition, Irwin, Homewood, Illinois, 1989.

22. Knight, Robert Lewis, Data Administration and its Role at Naval Supply Systems Headquarters, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1985.

23. McClure, Carma, CASE Is Software Automation, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

24. Ubois, Jeff, "IS Managers are Getting RADical in Developing in-house Applications," MacWeek, Vol. 6, No. 4, January 27, 1992.

25. Jacques, Trevor, "Prototyping Tools Ease Coding Burden," Computing Canada, Vol. 18, No. 4, February 17, 1992.

26. Jacques, Trevor, "From Code-and-Fix to 4GL's: why we need RAD," Computing Canada, Vol. 18, No. 2, January 20, 1992.

27. Martin, James, "CASE Tools To Play a Larger Role in IS Organizations," PC WEEK, July 2, 1990.

28. Martin, James, Information Engineering, Book II, Planning and Analysis, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1990.

29. Bloor, Robin, "CASE and Meta CASE: Powerful CASE Tools May be the Basis of a Wholly new Approach to the Development Environment," DBMS, Vol. 5, No. 1, January 1992.

30. Sittenauer, Chris, Olsem, Mike, Daich, Greg, Murdock, Daren and Janzen, Packer, Re-engineering Tools Report, March 1992, Software Re-engineering Tools Evaluation Project, Software Technology Support Center, Hill Air Force Base, Utah.

31. Request for Information (RFI) for Integrated Computer Aided Software Engineering (I-CASE), Standard Systems Center, Gunter AFB, Alabama, December 18, 1991.

32. Thomas, Ian and Nejmeh, Brian A., "Definitions of Tool Integration for Environments," IEEE Software, Vol. 9, No. 2, March 1992.

33. Chen, Minder and Norman, Ronald J., "A Framework for Integrated CASE," IEEE Software, Vol. 9, No. 2, March 1992.

34. "Cohesion: Your Open Advantage When Developing Software," Digital Equipment Corporation, 1992.

35. "Digitals Distributed Repository: Blueprint for Managing Enterprise-Wide Information," Digital Equipment Corporation, 1991.

36. RAD: Rapid Application Development Handbook, James Martin Associates Inc., 1850 Centennial Park Drive, Ruston, Virginia, USA, 1990.

37. Morris, Ed, Feiler, Peter, and Smith, Dennis, "Case Studies in Environment Integration," Technical Report CMU/SEI-91-TR13 ESD-91-TR-13, December 1991, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

38. Ruhl, Mary K., Gunn, Mary T., "Software Reengineering: A Case Study and Lessons Learned," CASE Strategies, Cutter Information Corporation, 37 Broadway, Arlington, Massachusetts.

39. Rothe, James E., "Re-engineering To Client/Server," Proceedings from Digital Consulting, Inc. National

Software Re-engineering & Maintenance Conference, San Jose, California, 10-12 August, 1992.

40. Jones, Capers, "Software Re-engineering and Measurement," Proceedings from Digital Consulting, Inc. National Software Re-engineering & Maintenance Conference, San Jose, California, 10-12 August, 1992.

41. Seymour, Patricia, "Critical Implementation Components and Cost Justification for Successful Software Re-engineering," Proceedings from Digital Consulting, Inc. National Software Re-engineering & Maintenance Conference, San Jose, California, 10-12 August, 1992.

42. Ulrich, William M., "Software Re-engineering: An Effective Combination of Methods & Tools," Proceedings from Digital Consulting, Inc. National Software Re-engineering & Maintenance Conference, San Jose, California, 10-12 August, 1992.

43. Chikofsky, Elliot, "Untying the spaghetti: an expert picks the tools." Datamation, Vol. 38, No. 9, April 15, 1992.

44. Jones, Capers, "Applying Total Quality Management (TQM) to Software," Software Productivity Research, INC., August 4, 1992.

45. "Ada Transition Research Project (Phase II) ASQB-GI-92-004," US Army Information Systems Engineering Command, Fort Huachuca, Arizona, April 1992.

46. "McCabe Tools Catalog: a comprehensive set of software analysis tools for UNIX, VMS, and DOS," McCabe & Associates, Columbia, Maryland.

47. Knowledgeware Inc., Redevelopment Products, Atlanta, Georgia, 1992.

48. "TI's IEF scores high for integration, benefits delivery," Computerworld, April 22, 1991.

49. Telephone conversation between Donald Griest of Knowledgeware Inc., and the author, 2 September 1992.

50. Datapro Information Services Group, Texas Instruments Information Engineering Facility, McGraw-Hill, 1991.

51. Telephone conversation between Jim Hawthorne, LTCOL, USA, Directorate of Defense Information/Information Technology, and the author, 4 September 1992.

52. Telephone conversation between Kay Jain of Texas Instruments, and the author, 04 September 1992.

53. Telephone conversation between Dr. Bill Curtis of the Software Engineering Institute at Carnegie-Mellon University, and the author, 25 August 1992.

54. "The Re-engineering Center: A Price Waterhouse Initiative," CASE STRATEGIES, Vol. 2, No. 8, August, 1990.

INITIAL DISTRIBUTION LIST

		NO. COPIES
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Professor Martin J. McCaffrey, Code AS/MF Department of Administrative Sciences Naval Postgraduate School Monterey, California 93943-5002	1
4.	Professor Tung Bui, Code AS/BD Department of Administrative Sciences Naval Postgraduate School Monterey, California 93943-5002	1
5.	LT Charles Jennings, USN 6508 Whitesburg Dr. Huntsville, Alabama 35802	1

Thesis
J466 Jennings
c.1 Re-engineering software systems in the Department of Defense using integrated computer aided software engineering tools.

Thesis
J466 Jennings
c.1 Re-engineering software systems in the Department of Defense using integrated computer aided software engineering tools.

DUDLEY KNOX LIBRARY
3 2768 00036349 3